KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
KOMPIUTERIŲ KATEDRA

Justinas Kuizinas

Daiktų interneto įrenginių autentifikavimas panaudojant blokų grandinės technologiją

Baigiamasis magistro darbas

Vadovas:
Prof. dr. Algimantas Venčkauskas

Kaunas, 2020

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
KOMPIUTERIŲ KATEDRA

Daiktų interneto įrenginių autentifikavimas panaudojant blokų grandinės technologiją

Baigiamasis magistro darbas
Informacijos ir informacinių technologijų sauga (kodas 6211BX008)

Atliko:
Justinas Kuizinas parašas
Vadovas:
Prof. dr. Algimantas Venčkauskas parašas
Recenzentas:
Doc. Rasa Brūzgienė parašas

Kaunas, 2020

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
Justino Kuizino
Informacijos ir informacinių technologijų sauga (kodas 6211BX008)

„Daiktų interneto įrenginių autentifikavimas panaudojant blokų grandinės technologiją“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

2020 m. Birželio 1 d.

Kaunas

Patvirtinu, kad mano **Justino Kuizino** baigiamasis projektas tema „Daiktų interneto įrenginių autentifikavimas panaudojant blokų grandinės technologiją“ yra parašytas visiškai savarankiškai, o visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

_____ _____

Kuizinas, Justinas. „Daiktų interneto įrenginių autentifikavimas panaudojant blokų grandinės technologiją“. Magistro baigiamasis projektas / vadovas prof. dr. Algimantas Venčkauskas; Kauno technologijos universitetas, Informatikos fakultetas, Kompiuterių katedra.

Studijų kryptis ir sritis: Informatikos inžinerija, informacijos ir informacinių technologijų sauga

Raktiniai žodžiai: daiktų internetas, išmanieji įrenginiai, blokų grandinės, autentifikacija

Kaunas, 2020. 92 p.

Santrauka

Išmaniųjų įrenginių saugi autentifikacija yra viena pagrindinių saugumo problemų daiktų interneto tinkluose. Esami darbai bei sprendimai, bandantys spręsti šią problemą, dažnai neapgalvoja saugaus autentifikacijos kanalo, patiki autentifikaciją centriniam serveriui taip sukurdami SPOF (angl. Single Point of Failure) arba neužtikrina pakankamai saugaus autentifikacijos srauto. Šiame darbe siūlomas blokų grandinių technologija paremtas DI įrenginių autentifikacijos metodas, kuris patobulina esamus metodus pridėdamas antrą autentifikacijos faktorių, užtikrindamas saugią komunikaciją AMQPS protokolu bei operuodamas privačiame „Hyperledger Sawtooth“ blokų grandinių tinkle. Pasiūlytame sprendime išmanieji įrenginiai suformuoja unikalų identifikatorių registracijos metu, kuris tampa globaliu identifikatoriumi sistemoje ir jo vardu išduodami TLS sertifikatai, įrenginys užregistruojamas žinučių brokerio serveryje ir įrašomas į blokų grandinę. Įrenginiui siunčiant žinutę saugiu kanalu AMQPS protokolu, atliekama pirminė autentifikacija žinučių brokeryje, kur patikrinamas įrenginio sertifikatas ir įrenginys identifikuojamas pagal sertifikato klientą (CN). Sėkmingai atlikus pirminę autentifikaciją žinutė ištransliuojama blokų grandinės tarpininko klientui, kuris atsakingas už antrinę autentifikacija blokų grandinės tinkle. Tik sėkmingai atlikus abu autentifikacijos žingsnius įrenginys yra sėkmingai autentifikuojamas.

Siekiant įsitikinti sprendimo patikimumu buvo sukurtas prototipas, įgyvendinantis aprašytą architektūrą. Sukurtoja aplinkoje atlikti kiekybiniai ir kokybiniai eksperimentiniai tyrimai ir įsitikinta pasiūlyto metodo efektyvumu ir saugumo charakteristikomis.

Kuizinas, Justinas. *Blockchain Based Authentication Method for IoT Devices* Master's thesis in Information and Information Technology Security / supervisor prof. Algimantas Venčkauskas. The Faculty of Informatics, Kaunas University of Technology.

Research area and field: Informatics Engineering, Information and Information Technology Security

Key words: internet of things, smart devices, blockchain, authentication

Kaunas, 2020. 92 p.

Summary

Secure authentication is one of the major problems in IoT (Internet of Things) networks. The past works and solutions that try to solve this problem but often do not secure communication channel between the device and main server, entrusts authentication to the central server thus creating SPOF (Single Point of Failure) or does not provide a sufficiently secure authentication. This paper proposes a method of authentication of IoT devices based on blockchain technology which enhances existing methods by adding a second authentication factor, ensuring secure communication using the AMQPS protocol and operating in a private Hyperledger Sawtooth blockchain platform. In the proposed solution smart devices during registration phase form a unique identifier which becomes the global identifier in the system. By using this identifier TLS certificates are issued on its behalf, the device is registered then on the message broker and lastly - smart device is registered on the blockchain state as well. While sending a message by the device over a secure channel using AMQPS protocol, initial authentication is performed by the message broker server RabbitMQ, where device's certificate is verified and the device is identified by the common name (CN) value which is stored in the certificate itself. Upon successful completion of the initial authentication the message is transmitted to the blockchain's intermediary client that is responsible for secondary authentication in the blockchain network. Only upon successful completion of both authentication stpes the device is successfully authenticated.

To ensure the reliability of the solution - a prototype was developed and implemented as according to the designed architecture. By conducting qualitative and quantitative researches the efficiency and security properties of proposed authentication method were approved.

Turinys

Iliustracijų sąrašas

Lentelių sąrašas

TERMINŲ IR SANTRUMPŲ ŽODYNAS

- Hakeris - kompiuterių ekspertas, siekiantis įsilaužti į sistemą ar kitaip ją pažeisti
- Wi-fi – Belaidžio ryšio technologija
- IPv6 – 6-os versijos interneto protokolas, naudojantis 128-ių bitų adresą, taip įgalinantis didesnį kiek adresų.
- Bluetooth – belaidžio ryšio gamybinė specifikacija, leidžianti keistis informacija tarp įrenginių.
- RS232 - duomenų perdavimo standartas, skirtas sujungti kompiuterį bei su juo susijusį įrenginį.
- Vienos plokštės kompiuteris – ant elektroninės grandinės plokštės veikiantis kompiuteris, galintis atlikti beveik visas normalaus kompiuterio funkcijas.
- CoAPP (angl. Constrained Application Protocol) – Ribotas Programos Protokolas
- MQTT (angl. Message Queue Telemetry Transport) – Pranešimo Eilės Telemetrijos Transportas
- XMPP (angl. Extensible Messaging and Presence Protocol) – Išplečiamas Pranešimų ir Dalyvavimo Protokolas
- REST (angl. Representational State Transfer) – Reprezentacinis Struktūros Perdavimas
- AMQP (angl. Advanced Message Queuing Protocol) – Pažengęs Pranešimo Eilės Protokolas
- WSN (angl. Wireless Sensor Networks) – Belaidžių Jutiklių Tinklas
- DLT (angl. Distributed Ledger Technology) – Paskirstytų Duomenų Technologija
- PoW (angl. Proof of Work) – aprašom mechanizmą, kuris leidžia sukurti naują bloką tik kas tam tikrą laiko intervalą.
- PoS (angl. Proof of Stake) – aprašo mechanizmą, kai kuo didesnę dalį turi blokų grandinės tinkle naudotojas, tuo daugiau įtakos jis turi, pavyzdžiui, kuriant bloką.
- Dapp (angl. Decentralized application) – tai programa, kuri dirba ant decentralizuotų tinklų.
- PoET (angl. Proof of Elapsed Time) – konsensuso mechanizmas, kuris yra efektyvesnė PoW forma, pašalinanti "kasimo" procesą ir pakeičianti jį atsitiktine imčių laikmačio sistema.
- PBFT (angl. Practival Byzantine Fault Tolerance) – mechanizmas, kuris reikalauja 2/3 sistemos narių sutikimo patvirtinant sprendimą.
- ECDH (angl. Elliptic-curve Diffie-Hellman) – anonimiškas raktų susitarimo protokolas
- SASL (angl. Simple Authentication and Security Layer) – paprastas autentifikavimo bei saugumo sluoksnis

- TLS (angl. Transport Layer Security) – kriptografinis protokolas, numatantis apsaugotą duomenų perdavimą tarp mazgų
- BPMN (angl. Business Process Model and Notation) – tai grafinė reprezentaciją specifikuoti verslo procesus verslo procesų modelyje
- RSA – tai viena pirmųjų viešojo rakto kriptosistemų, plačiai naudojama saugiam informacijos perdavimui
- ECDSA (angl. Elliptic Curve Digital Signature Algorithm) – elipsinės kreivės skaitmeninio parašo algoritmas
- EdDSA (angl. Edwards-curve Digital Signature Algorithm) – tai skaitmeninio parašo schema, kurioje naudojamas "Schnorr" parašo variantas ir pagrįstas "Twisted Edwards" kreivėmis.
- PSK (angl. Pre-Shared Key) – tai bendra paslaptis, kuri prieš tai buvo naudojama saugiame kanale.
- PnP (angl. Plug and Play) – sistemų projektavimų metodas, kai sisteminis komponentas yra visiškai nepriklausomas nuo platformos ir gali būti tiesiog prijungiamas ar išjungiamas, nesutrikdant visos sistemos darbo
- P2P (angl. Peer-to-Peer) – tinklo modelis, kuriama komunikacija tarp klientų vyksta tiesiogiai.
- SPOF (angl. Single Point of Failure) – pavienis pažaidos taškas
- DoS (angl. Denial of Service) bei DDoS (angl. Distributed DoS) – paslaugos trikdymo ataka bei paskirstyta paslaugos trikdymo ataka

Įvadas

Darbo problematika ir aktualumas:

Nesaugus daiktų interneto įrenginių autentifikavimas. Daiktų interneto įrenginių tinklai įgauna vis didesnę reikšmę bei tampa vis plačiau aptarinėjama tema dėl sparčiai didėjančios paklausos bei panaudojimo masto. Tačiau iškyla problema - kaip sujungti tokius įrenginius į tinklą, kuriame jie galėtų dalintis informacija, tačiau kartu ta informacija būtų apsaugota nuo kibernetinių atakų ir išliktų privati?

Darbo sritis - informacijos bei informacinių technologijų saugos sritis, daiktų interneto įrenginių komunikacijos problema, blokų grandinės panaudojamumo analizė saugumo atžvilgiu bei saugaus autentifikavimo matodo sukūrimas ir tyrimas.

Darbo tikslas ir uždaviniai: Tikslas: sukurti saugesnį autentifikavimo metodą, kuris būtų paremtas blokų grandinės technologija. Uždaviniai:

- išsiaiškinti pagrindines daiktų įrenginių autentifikavimo problemas;
- pasiūlyti saugesnį DI įrenginių atpažinimo metodą panaudojant blokų grandinių technologiją, pridedant antrą autentifikavimo faktorių;
- suprojektuoti siūlomą saugesnį autentifikacijos sprendimą;
- sukurti pasiūlyto autentifikavimo metodo prototipą;
- pateikti sukurto sprendimo tyrimo įvertinimą.

Darbo rezultatai ir jų svarba: Pagrindiniai darbo rezultatai:

- išskirtos pagrindinės daiktų interneto įrenginių autentifikavimo problemos. Tai padės apibrėžti būtinas sprendimo charakteristikas;
- pasiūlytas įgyvendinamas daiktų įrenginių autentifikacijos sprendimas paremtas blokų grandinių technologija, kuris užtikrintų saugesnį įrenginių autentifikavimą;
- pateikti kokybiniai bei kiekybiniai tyrimo rezultatai, kurie padėtų įvertinti sukurto sprendimo saugumo charakteristikas bei palyginti su egzistuojančiais sprendimais.

Darbo struktūra: Darbą sudaro 4 pagrindinės dalys:

- analizės skyrius „Daiktų interneto saugumo ir įrenginių autentifikavimo problemos“ - apžvelgiamas daiktų internetas bei blokų grandinės, analizuojami eksperimentiniai metodai, išskiriamos autentifikavimo problemos;
- sprendimo projektavimas „Daiktų interneto įrenginių autentifikavimo modelis panaudojant blokų grandinės technologiją“ - sukuriamas sprendimas, apibrėžiama architektūra bei pagrindinės metodo savybės;
- prototipo kūrimas „Daiktų interneto įrenginių autentifikavimo sistemos prototipas“ - aprašomas praktinis teorinio sprendimo įgyvendinimas;
- tyrimas „Daiktų interneto įrenginių autentifikavimo sistemos prototipo eksperimentinis tyrimas“ - pateikiamas kokybinis bei kiekybinis sukurto sprendimo tyrimas.

Darbą sudaro 92 puslapiai.

1 Daiktų interneto saugumo ir įrenginių autentifikavimo problemos

Per pastaruosius dešimtmečius buvo skiriama daug dėmesio bei lėšų tyrimams siekiant įvertinti jutiklių bei komunikacijų technologijas. Tuo metu jutiklinės sistemos buvo linkusios plėtotis lokalizuotose taikomosiose ir technologinėse srityse. Pirmieji rezultatai parodė, kad jutiklių duomenys gali būti naudojami įvairiems tikslams, jei tik būtų galima sukurti dalijimosi bei bendravimo priemones. Šį siekį buvo siekiama panaudoti terminui „daiktų internetas“, kuris pirmą kartą viešai paskelbtas 1999 metais autoriaus Kevino Ashtono. Nuo pirmo šio apibrėžimo pavartojimo – technologija smarkiai plėtėsi ir šiuo metu yra plačiausiai aptarinėjama tema.

Šiame skyriuje atliekamas bei pateikiamas tyrimas, kurio metu siekiama išsiaiškinti kas yra daiktų internetas (toliau DI) bei jo saugumo problemas. Taip pat būtina išsiaiškinti DI įrenginių komunikavimo ir autentifikavimo metodus, juos išanalizuoti, rasti komunikavimo protokolus bei eksperimentinius metodus. Galiausiai – įvardinti DI įrenginių identifikavimo ir autentifikavimo metodus.

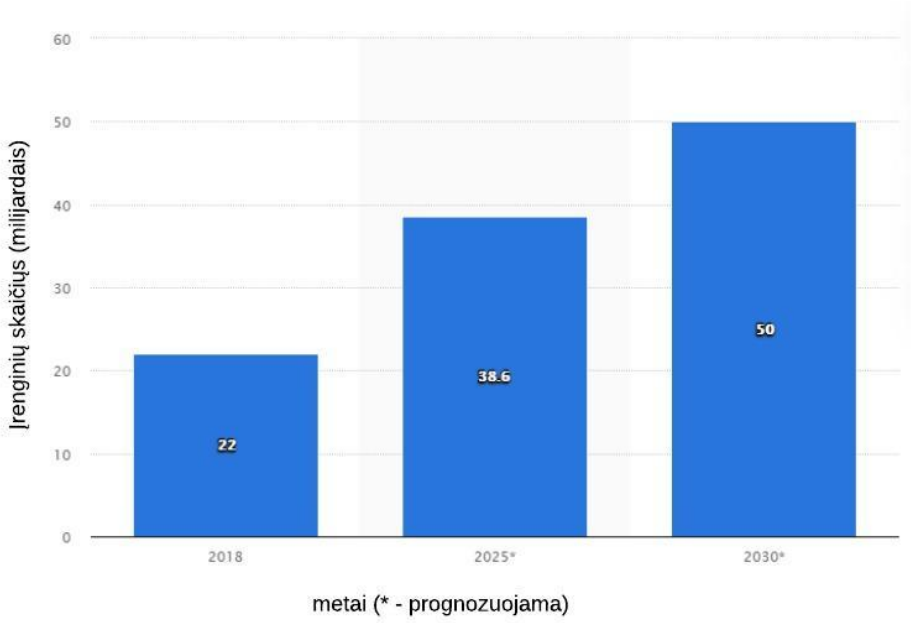
Po daiktų interneto analizės dalies pateikiama blokų grandinės technologijos analizė: blokų grandinių technologijos aptarimas, principų išsiaiškinimas, šia technologija paremtų autentifikacijos metodų bei platformų suradimas ir analizė. Pagrindinis uždavinys šio skyriaus – ištyrus esamus metodus išskirti pagrindines autentifikavimo problemas ir pasiūlyti saugesnį DI įrenginių autentifikavimo sprendimą, kuris būtų paremtas blokų grandinės technologija.

1.1 Daiktų internetas ir jo saugumo problemos

Daiktų internetas (angl. *Internet of things – IoT*) – yra įvairių fizinių prietaisų tinklas, kuriame jie gali dalintis informacija, o dauguma autorių literatūroje šią sąvoką įvardina kaip pagrindinį ketvirtosios pramoninės revoliucijos komponentą [4]. Laima Zalieckaitė su Raimundu Žilinsku savo darbe „Daiktų interneto technologijos taikymo versle nauda ir rizika“ [5], apibendrinę įvairias daiktų interneto sąvokas teigia, jog „[...] daiktų internetas yra tinklas, kuriame fiziniai objektai sujungti tarpusavyje, žmonių ir objektų bendravimas vyksta per fizinius objektus, o valdymas vykdomas virtualiai.“ – taip įtraukdami ir žmonių rolę į daiktų interneto sąvoką. Užsienio autoriai daiktų internetą apibūdina gan panašiai, pavyzdžiui, Al-Fuqaha, A. Guizani ir kiti autoriai savo darbe [8] apibūdina šią sąvoką kaip technologiją, kuri leidžia fiziniams objektams matyti, klausyti, mąstyti ir atlikti kitus darbus priverčiant juos visus bendrai komunikuoti, dalintis informacija ir koordinuoti sprendimus. Tame pačiame šaltinyje pateikiami duomenys, jog 2010 metais DI įrenginių skaičius pralenkė žmonių skaičių žemėje, be to, IPv6 adresai tapo būtinybe siekiant patenkinti naudotojų paklausą išmaniesiems įrenginiams.

„Statista“ surinktais duomenimis, pavaizduotais 1 paveikslėlyje, 2018m sujungtų DI įrenginių skaičius pasaulyje siekė 22 milijardus. Taip pat prognozuojama, jog 2025 metais metais šis skaičius turėtų padidėti iki beveik 37 milijardų ir iki 50 milijardų 2030 metais, o taip didėjant tokių įrenginių skaičiui, saugumas turėtų būti išskiriamas kaip vienas pagrindinių faktorių.

Tuo tarpu „Forbes“ tinklaraštyje Jacobas Morganas savo straipsnyje teigia, jog netolimoje ateityje pagrindinė taisyklė bus „viskas, ką galima sujungti, bus sujungta“ [6], taip išryškindamas vis didėjantį žmonių susidomėjimą DI ir galimą šios technologijos perspektyvą. Taip pat šiame straipsnyje saugumas išskirtas kaip viena didžiausių ir potencialiausių problemų, su kuriomis susi-



1 pav. Sujungtų DI įrenginių skaičius pasaulyje milijardais

duriama daiktų internetui užimant vis didesnę vietą žmonių kasdieniniame gyvenime. Pagrindinės išskiriamos priežastys įvardinamos kaip nesuvaldomas įrenginių kiekio augimas, kuris natūraliai reikš didesnį informacijos kiekio laikymą tokiuose įrenginiuose, taip pritraukiant įvairiausių kenkėjiškai nusiteikusių naudotojų dėmesį.

Dar vieną problemą iškėlė J. Singh'as, T. Pasquier bei kiti autoriai darbe [20], kuriame teigiama, jog taip greitai vystantis daiktų internetui nėra pakankamai dėmesio skiriama galimiems saugumo iššūkiams atpažįstant tam tinklui priklausančius įrenginius, todėl devintasis jų apsvarstymas yra pavadintas „Identifying *Things*“, o lietuviškai išvertus reikštų „Atpažįstant daiktus". Taip autoriai išskiria išmaniųjų tinklo įrenginių atpažinimą bei autentifikavimą kaip vienus kertinių veiksnių, kurie privalo būti apsaugomi. Šaltinio autoriai teigia, jog toks atpažinimo mechanizmas galėtų būti pateikiamas kaip atskira posistemė, kuri būtų atsakinga už daikto autentifikaciją, identifikaciją ir priskyrimą specifinei grupei (autorizacija).

„Nelegalaus įrenginio prieigos saugumo problema“ – taip įvardinama Q. Jing'o bei J. Lu darbe [9] problema, kuri taip pat apibūdina identifikavimą bei autentifikavimo problemą daiktų interneto tinkle. Autoriai teigia, jog kiekvienas įrenginys DI tinkle privalo turėti galimybę patvirtinti kito įrenginio tapatybę bei komunikuoti tik su patvirtintais tinklo daiktais. Šiame šaltinyje teigiama, jog šią problemą gali išspręsti autentifikacija ir autorizacija, tačiau išmaniųjų įrenginių tinkluose tokie sprendimai turi būti gerai apgalvoti ir ištirti, siekiant apsaugoti išmanųjį įrenginį nuo neleistino prisijungimo ar jautrios informacijos atskleidimo.

Apibendrinant daiktų internetą bei daiktų interneto saugumo problemas galima teigti, jog ši technologija vystosi ir plėtojasi dideliu greičiu, todėl saugumo tema šiai technologijai yra labai svarbi. Išnagrinėjus įvairių autorių pateiktus darbus buvo pastebėta, jog pagrindinės daiktų interneto saugumo problemos yra išskiriamos kaip informacijos kiekio laikymas tokiuose tinkluose bei išmaniųjų tinklo įrenginių atpažinimas bei autentifikavimas.

1.2 Daiktų interneto įrenginių komunikavimo ir autentifikavimo metodai

Siekiant visus prietaisus sujungti į vieną tinklą ir leisti jiems komunikuoti tarpusavy, iškyla būtinybė naudoti įrenginių komunikavimo protokolą, kuris apibrėžtų kaip įrenginiai bendraus vienas su kitu. Šis skyrius skirtas aptarti galimus ir naudojamus daiktų interneto įrenginių komunikavimo protokolus bei išanalizuoti veikiančias schemas ir eksperimentinius metodus. Vėliau – pateikiami ir išnagrinėjami galimi daiktų interneto identifikavimo ir autentifikavimo metodai taip siekiant išsiaiškinti kaip galima atpažinti įrenginius tokiuose tinkluose ir pateikti apibendrintas pastebėtas saugumo spragas.

1.2.1 Daiktų interneto įrenginių komunikavimo protokolai ir eksperimentiniai metodai

Visi protokolai skirstomi į 2 grupes – belaidžiai ir laidiniai. Išmanieji įrenginiai, komunikuojantys greitesniu belaidžiu ryšiu, atveria galimybę sistemą pažeisti internetiniams įsilaužėliams (angl. hacker), o tiesiogiai laidais sujungtomis linijomis bendraujantys įrenginiai sukuria saugesnį ryšį, tačiau apriboja nuotolį laidais, kurie nesuteikia didelio patogumo. Pagrindiniai plačiai naudojami bendravimo protokolai [12]:

- *Bluetooth Mesh* – plačiai naudojamas aukštų dažnių protokolas, bendraujantis *Bluetooth* ryšiu.
- *ZigBee* – aukšto lygio belaidis protokolas, veikiantis tinklelio (angl. mesh) principu ir plačiai paplitęs DI bei išmaniųjų namų tinkluose.
- *Z-Wave* – labai panašus į *ZigBee* protokolą, taip pat operuojantis tinklelio principu, naudojantis mažai energijos reikalaujančiomis radijo bangomis ir paprastesnis vystymo etape [1].
- *6LowPAN* [2] – pagrindinis šio protokolo skirtumas nuo kitų yra tas, jog šis protokolas yra interneto protokolas ir palaikantis IPv6 adresus, kurie suteikia galimybę daugybei įrangos turėti savo atskirą adresą internete.
- *Thread* [3] – gan naujas protokolas, paremtas IPv6 bei 6LowPAN standartais, skirtas daiktų interneto naudojimui.

Tokius tinklo lygio protokolus apibendrina P.Sethi ir R. Sarangi savo darbe [12], tačiau tinklo protokolai apibrėžia kaip įrenginiai bendraus tinkle. Todėl autoriai išskiria DI architektūrą į tris pagrindinius sluoksnius, kurie pavaizduoti 2 paveikslėlyje:

- Suvokimo sluoksnis – šis sluoksnis yra fizinis, kuris atsakingas už jutiklių darbą bei kitų išmaniųjų daiktų atpažinimą.
- Tinklo sluoksnis – atsakingas už DI tinklo įrenginių sujungimą su serveriais. Taip pat atsakingas už jutiklių informacijos perdavimą tinklu bei komunikaciją naudojant anksčiau išvardintus protokolus.
- Programos (informacijos) sluoksnis – šis sluoksnis skirtas perduoti programos teikiamas paslaugas naudotojui, be to apibrėžia kur gali DI būti naudojamas, t.y., išmanieji namai, medicina ir kiti.

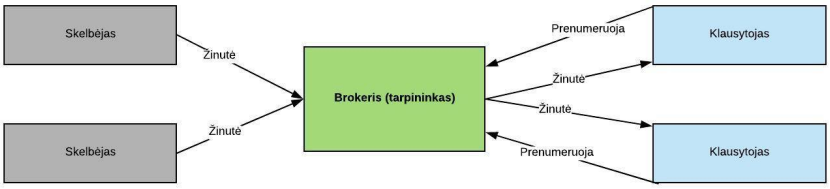
2 pav. DI architektūros sluoksniai

Programos sluoksnyje veikia jau kiti protokolai, kurie atsakingi už informacijos struktūrą bei perdavimą belaidžiu ryšiu. V. Karagiannis savo darbe apie programos sluoksnio protokolus skirtus DI [10] išskiria šiuos dažniausiai naudojamus protokolus:

- CoAPP(angl. Constained Application Protocol – ribotos programos protokolas) – sinchroniškų užklausų bei atsakų protokolas, kuris skirtas ribotų resursų įrenginiams. Šis protokolas sukurtas specifiškai daiktų internetui ir suderinamas su HTTP, tačiau neturi jokių saugumo funkcijų.

- MQTT(angl. Message Queue Telemetry Transport – pranešimo eilės telemetrijos transportas) – sukurtas IBM ir skirtas lengvoms komunikacijoms tarp įrenginių, šis protokolas yra paremtas skelbimo / prenumeravimo principu (žr. pav. 3), kai įrenginiai nelaukia atsako po užklausos, o tiesiog užsiprenumeruoja tam tikros grupės naujienas ir iškart gauna informaciją, kai ji yra paskelbiama. Kaip teigiama šio protokolo oficialiame tinklapyje [21] – tai daiktų interneto jungties protokolas ir buvo sukurtas kaip neįtikėtinai lengvas pranešimų transportavimo protokolas. Verta paminėti, jog šį protokolą naudoja garsaus saityno „Facebook Messenger“ programėlė [22].

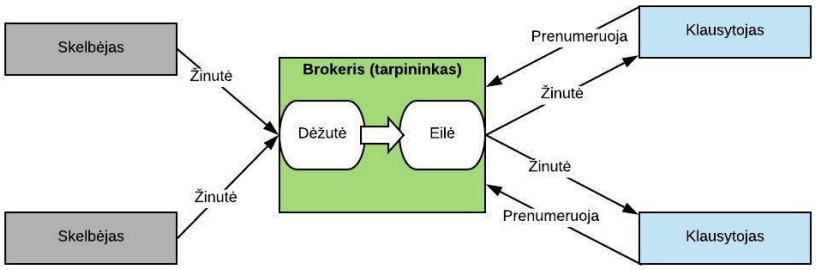
- REST(angl. Representational State Transfer – reprezentacinis struktūros perdavimas) – labiau ne protokolas, tačiau architektūrinis komunikavimo formatas. REST veikia su HTTP protokolu ir daug informacijos perduoda per užklausos antraštę, kuri nusako koks bus duomenų formatas, o užklausos vyksta sinchroniniu užklausos / atsako būdu. Tačiau tarp DI



3 pav. MQTT protokolo modelis

įrenginių naudojamas vis rečiau, kadangi tokių HTTP užklausų apdorojimas ir implementacija reikalauja resursų, kurių apribotų resursų įrenginiai neturi pakankamai.

- AMQP(angl. Advanced Message Quieuing Protocol – pažengęs pranešimo eilės protokolas) – asinchroniškų skelbimo/prenumeravimo žinučių protokolas(žr. 4 pav.), kuris dėmesio susilaukė po įvairių įrodytų tyrimų, kurie parodė, jog AMQP yra pajėgus išsiųsti ženkliai didesnį skaičių žinučių per sekundę nei anksčiau aprašytas REST protokolas [24]. Kaip teigiama oficialiame tinklalapyje [23], AMQP protokolas siekia saugumo, patikimumo, atvirumo, standartiškumo bei sąveikiškumo. Šio protokolo modelis yra gan panašus į anksčiau minėto MQTT, tačiau pagrindinis skirtumas yra tai, jog AMQP žinutės yra saugomos eilėse, kurios yra saugomos bei skelbiamos, todėl reikalauja didesnio tinklo pralaidumo bei resursų kiekio tinklo įrenginiuose.



4 pav. AMQP protokolo modelis

- Websockets – šio protokolo atveju klientas su serveriu inicializuoja sesiją ir vadinamąjį „rankų paspaudimą“, po kurio žinutės keliauja į abi puses asinchroniškai. Tačiau DI programose tai naudojama rečiau, kadangi šis protokolas reikalauja pakankamai didelių resursų, kurių, kaip jau minėta anksčiau, kai kurie DI įrenginiai neturi pakankamai.

Nitino Naiko darbas apie efektyvių komunikavimo protokolų pasirinkimą DI sistemoms [26] išskiria MQTT, CoAP bei AMQP protokolus kaip pagrindinius pasirinkimus kuriant daiktų interneto sistemą. Siekiant detalizuoti savo temą, autorius atlieka santykinę analizę su kituose šaltiniuose pateikiama informacija apie išsirinktas technologijas, pavyzdžiui: energijos suvartojimas prieš resursų reikalavimą, pralaidumas prieš pranešimų vėlavimą ir saugumą prieš prieinamumą. Išvadose autorius apibendrina darbą ir teigia, jog MQTT bei AMQP protokolai labiausiai tinkami DI srityje, o CoAPP bei HTTP yra labiau standartizuoti modeliai. Tuo tarpu J. Luzuriaga, M. Perez ir kiti autoriai darbe [28] palygina MQTT ir AMQP protokolus nestabiliuoze ir mobiliuose

	MQTT	AMQP
Architektūra	Klientas-Brokeris	Klientas-Brokeris/Klientas-Serveris
Antraštės dydis	2 baitai	8 baitai
Transporto protokolas	TCP	TCP
Saugumas	TLS	TLS, SASL
Abstrakcija	skelbimas-prenumeravimas	skelbimas-prenumeravimas/ užklausa-atsakymas
Žinučių eilės	Ne	Taip

1 lentelė. AMQP ir MQTT protokolų palyginimas

	RabbitMQ	ActiveMQ	IronMQ
Decentralizuota	Ne	Ne	Taip
Tranzakcijos	Taip	Taip	Ne
Maršrutizavimas	Taip	Taip	Ne
Protokolai	STOMP, MQTT, AMQP	XMPP, REST, MQTT, AMQP ir dar daugiau	HTTPS
Aukšto pasiekiamumo mechanizmas	Automatinis atsistatymas (failover)	Automatinis atsistatymas (failover)	Debesų infrastruktūra
Prenumeravimo-skelbimo palaikymas	Taip	Taip	Taip
Atviro kodo	Taip	Taip	Ne
Autentifikacijos mechanizmai	AMQPLAIN, x.509, PLAIN	PLAIN, JAAS	OAauth

2 lentelė. Žinučių brokerių palyginimas

tinkluose praktiškai atlikdami tyrimą su klientu, serveriu bei dviem prieigos taškais. Rezultatuose autoriai pateikia, jog abu protokolai užtikrina žinučių perdavimą be nuostolių net nestabiliame tinkle ir teigia, jog AMQP 1.0 versija leidžia užtikrinti saugesnį ryšį dviem sluoksniais [29]:

- SASL (angl. Simple Authentication and Security Layer) – atviro teksto naudotojo prisijungimo vardo bei slaptažodžio autentifikacijos bei saugumo mechanizmą, kuris skirtas naudoti kartu su žemesnio sluoksnio paslaugomis (pavyzdžiui, TLS].
- TLS (angl. Transport Layer Security) – kriptografinis protokolas, užšifruojantis SASL mechanizmo duomenis.

Lentelėje 1 išskirti pagrindiniai faktoriai, lyginant AMQP ir MQTT protokolus. Architektūra panaši, tik AMQP turi galimybę komunikaciją atlikti ir užklausos-atsakymo būdu, taip pat AMQP turi 4 kartus didesnį antraštės dydį. Aktualiausio šio darbo tematikoje skirtumai – tai saugaus informacijos pernešimo užtikrinimas, kurį tiek MQTT, tiek AMQP užtikrina integruojant TLS, tačiau AMQP papildomai suteikia galimybę naudoti SASL, taip išplečiant saugumo galimybęs naudojant AMQP protokolą.

Apžvelgus programos sluoksnio protokolus ir išskyrus DI įrenginių komunikacijai tinkamiausius – būtina apžvelgti galimus žinučių brokerius. Abhijith Reddy savo straipsnyje išskiria vienus pagrindinių kandidatų renkantis tarp skirtingų žinučių brokerių [11]: „RabbitMQ“, „ActiveMQ“ ir „IronMQ“. Lentelėje (2) pateikti analizės rezultatai ir palyginimas skirtingų žinučių brokerių. „RabbitMQ“ tai atviro kodo, „Pivotal Software“ kuriamas pranešimų tarpininkavimo serveris, kuris pirmiausia buvo sukurtas kaip įrankis skirtas AMQP protokolo komunikacijai, o vėliau buvo pritaikytas ir MQTT bei STOMP protokolo palaikymui, įgyvendina tranzakcijų funkcionalumą bei žinučių maršrutizavimą remiantis skirtingais metodais, palaiko automatinį atsistatymą (angl. failover) taip užtikrindamas aukštą pasiekiamumą, taip pat palaiko prenumeravimo-skelbimo abstrakciją, tačiau yra centralizuotas serveris. „ActiveMQ“ kaip ir „RabbitMQ“ yra atviro kodo žinučių brokeris, kurį kuria ir prižiūri „Apache“. Jį lyginant su „RabbitMQ“ pagrindinis skirtumas yra tik palaikomų protokolų skaičius, nes „ActiveMQ“ palaiko daugybę kitų programos sluoksnio protokolų. „IronMQ“ tuo tarpu yra uždaro kodo verslo lygio sprendimas, kuris žinučių brokerio funkcijas atlieka debesyse, taip užtikrindamas decentralizuotą sprendimą bei aukšto pasiekiamumo užtikrinimą. Tačiau pagrindiniai trūkumai tai tranzakcijų ir maršrutizavimo nepalaikymas bei komercinė licenzija, kuri gali brangiai kainuoti priklausomai nuo naudojamo plano. Dar vienas svarbus aspektas renkantis žinučių brokerį – autentifikacijos mechanizmai. „RabbitMQ“ siūlo atvirus *AMQPLAIN* ir *PLAIN* metodus, kurie autentifikuoja naudotoją pagal prisijungimo vardą bei slaptažodį, o x.509 autentifikacijos metodas pasiūlo galimybę autentifikuoti klientą pagal sertifikatą taip nereikalaujant slaptažodžio, o naudotojo vardą pasiimant iš sertifikato vardo – CN (angl. Common Name). „ActiveMQ“ taip pat siūlo atvirą *PLAIN* metodą bei papildomai JAAS (angl. Java Authentication and Authorization Service) servisą, kuriuo pasinaudojus galima aprašyti reikiamą autentifikacijos logiką ir ją prijungti prie „ActiveMQ“ autentifikacijos varikliukų. Tuo tarpu „IronMQ“ naudoja *OAuth* mechanizmą autentifikacijai, kuris yra paremtas užklausų ženklų (angl. tokens) principu: žetono gavimas su prisijungimo duomenimis, žetono įdėjimas į užklausos antraštę, žetono ištrynimas pasibaigus sesijai.

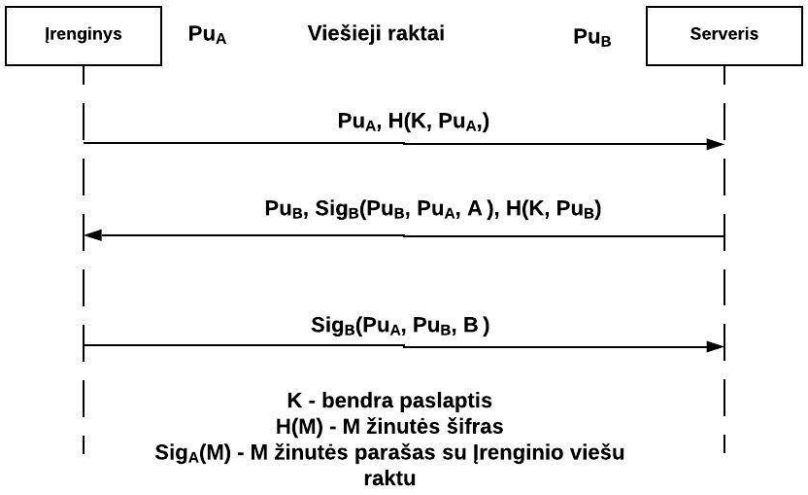
Apibendrinant šį skyrių galima teigti, jog daiktų interneto architektūra yra skirstoma į tris lygius: suvokimo, tinklo ir programos sluoksnius. Tinklo ir programos sluoksniuose operuoja įvairūs protokolai, tačiau renkantis saugų ir efektyvų daiktų interneto įrenginių komunikacijos protokolą – įvairūs autoriai išskiria AMQP kaip geriausią pasirinkimą dėl patikimumo, saugumo savybių bei plataus panaudojamumo. Renkantis žinučių brokerį pagrindiniai kandidatai yra „RabbitMQ“ bei „ActiveMQ“ servisai.

1.2.2 Daiktų interneto įrenginių identifikavimo ir autentifikavimo metodai

Kaip jau išsiaiškinta ankstesniame skyriuje 1.2.1 – daiktų interneto tinkle įrenginiai gali būti sujungiami laidais arba belaidžiu ryšiu, tačiau dauguma tinklų sujungiami belaidžiu ryšiu, kadangi tai yra lankstesnis būdas prijungti įrenginį į tinklą, o tokie tinklai dažniausiai vadinami WSN (angl. Wireless Sensor Networks – belaidžių jutiklių tinklas). Tačiau natūralus belaidžio kanalo būdas atidaro kelią ir piktavališkiems naudotojams lengviau gauti tinklo informaciją ar neleistinai prisijungti prie tinklo. Vienas pagrindinių šios problemos sprendimų, paminėtų ankstesniame skyriuje(1.1) – įrenginių autentifikacija ir identifikacija.

Vienas iš siūlomų autentifikacijos metodų, pasiūlytas F. Santos'o ir N. Vun'o [27] yra ECDH bendro rakto panaudojimas atpažįstant tinklo įrenginį. Vienas pagrindinių privalumų yra tai, jog

šis sprendimas gan paprastai sukuriamas, operuoja Wi-Fi, belaidžio ryšio technologijos, tinklu ir yra pakankamai saugus. Įrenginio autentifikacija, parodyta 5 paveikslėlyje, paremta bendro slapto rakto žinojimu, kuris naudojamas užšifruojant žinutę. Šio sprendimo pagrindiniai trūkumai būtų: nepatogus ir neautomatizuotas naujo įrenginio įvedimo į tinklą procesas, nepritaikytas apribotų resursų įrenginiams, neišbandytas sprendimas su didesniu kiekiu įrenginių ir neatlikta analizė.



5 pav. Viešo rakto autentifikacija

Modernesnis ir saugesnis autentifikacijos metodas pasiūlytas X. Li, J. Niu ir kitų autorių darbe [13], kuriame analizės metu autoriai apibrėžia, jog RSA paremti sprendimai nelabai efektyvūs, kadangi reikalauja daug vietos ir resursų, kurių dauguma mažų jutiklių ar įrenginių neturi. Autorių pasiūlytas sprendimas turi kelias fazes: registracija ir autentifikacija. Registracijos metu įrenginiams, pagal unikalų jų identifikatorių, yra apskaičiuojamas vienakryptės maišos šifras (raktas), kuris kartu su įrenginio identifikatoriumi yra grąžinamas įrenginiui, o į centrinio serverio duomenų bazę įterpiamas įrenginio identifikatorius. Autentifikacijos atveju, pavyzdžiui, naudotojui norint peržiūrėti jutiklio informaciją, naudotojas autentifikuojamas ne tik centrinio serverio, tačiau ir įrenginio tokia eilės tvarka:

1. Naudotojas pateikia savo prisijungimo duomenis bei biometrinę informaciją ir užsifravęs kartu su jutiklio paslaptimi – siunčia į centrinį serverį;
2. Centrinis serveris gavęs užklausą iššifruoja duomenis turimu slaptu raktu bei patikrina vientisumo kodą. Jeigu visi patikrinimai buvo sėkmingi – užklausa yra nukreipiama į prašomą jutiklį;
3. Jutiklis šiuo atveju taip pat atlieka integrumo tikrinimą su savo turima paslaptimi ir atsiunčia sesijos rakto atsakymą centriniui serveriui, jeigu patikrinimas pavyko;
4. Centrinis serveris gavęs atsakymą – perduoda jį naudotojui;
5. Naudotojas gavęs sesijos rakto atsakymą jau yra autentifikuotas ir gali bendrauti su jutikliu.

Šis sprendimas užtikrina anonimiškumą, autentifikaciją tiek iš naudotojo pusės, tiek iš įrenginio, atsparus atkartojimo, apsimetinėjimo ir įrenginio perėmimo atakoms. Tačiau vienas pagrindinių trūkumų šios sistemose yra tai, kad autentifikacija negali įvykti, jeigu yra pažeidžiamas pagrindinis serveris.

1.3 Blokų grandinių technologija

Dar visai neseniai pasaulinę rinką sudrebino kriptovaliutos. „Bitcoin“, dažnai vadinama kaip pirmoji kriptovaliuta, susilaukė didžiulės sėkmės, kai jau 2016-ais metais jos rinkos vertė pasiekė 10 milijonų dolerių. Tačiau dauguma žmonių žino tik sąvoką *kriptovaliuta*, tačiau nežino *blokų grandinių* reikšmės.

Šiame skyriuje pagrindis keliamas tikslas yra išsiaiškinti blokų grandinių principus, taikymo sritis. Vėliau būtina išanalizuoti blokų grandinės technologija paremtus autentifikacijos metodus, ką jie suteikia sistemai, saugumo ypatumus. Galiausiai – pateikti blokų grandinių platformas, jų palyginimą bei taikymo sritis. Viską apibendrinant bus pateikiamos išvados ar blokų grandinės gali išspręsti daiktų interneto saugumo problemas.

1.3.1 Blokų grandinių principai

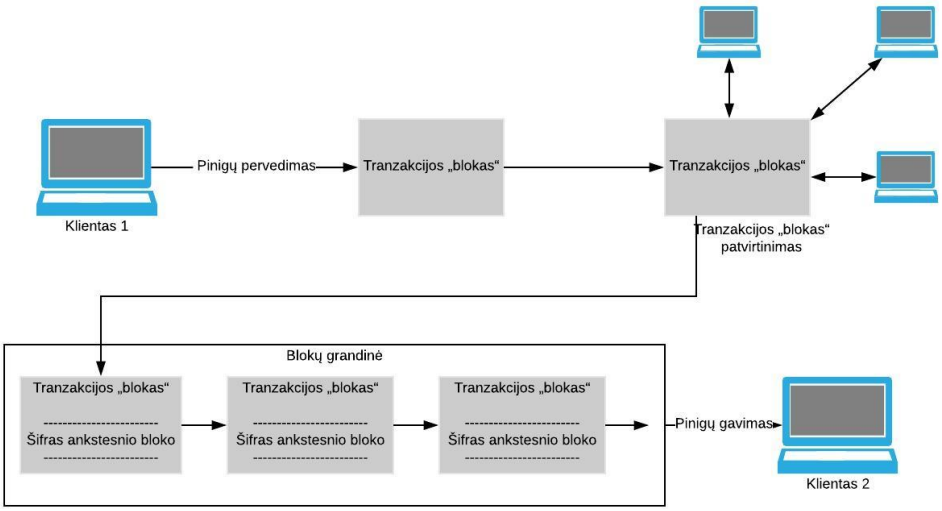
Blokų grandinės – tai pagrindinis mechanizmas kiekvienai kriptovaliutai. Šio mechanizmo pagrindinė technologija yra paskirstytų duomenų technologija (angl. DLT – Distributed Ledger Technology). DLT tai kartotinių, bendrų ir sinchronizuotų skaitmeninių duomenų sutarimas, taigi nėra vienos vietos, kurioje duomenys būtų saugojami – jie yra tinkle ir kiekvienas tinklo dalyvis žino informaciją apie pripažintą informaciją. Duomenys saugojami vadinamose *tranzakcijose*, kurios sudedamos į bloką iškart po tinklo dalyvių pripažinimo, jog ši tranzakcija yra validi. Kai tik informacija atsiranda bloke – ji negali būti koreguojama ar ištrinama. Blokų grandinės veikimą atspindi paveikslėlis 6, kuriame:

1. *Klientas 1*, norėdamas pervesti pinigus, sukuria tranzakcijos bloką, kuriame yra visa informacija apie mokėjimą
2. sukurtas blokas yra paskelbiamas visame tinkle, pranešant, jog kitas grandinės blokas bus būtent šis
3. gavus patvirtinimą iš tinkle esančių dalyvių – blokas atsiranda grandinėje, taip pat išsaugant informaciją apie prieš tai esantį bloką
4. galiausiai – pinigai pereina pas *Klientas 2*.

Būtina paminėti, jog tranzakcijos blokas saugo prieš jį einančio bloko šifruotą reikšmę, taip parodydamas savo poziciją, o pats pirmasis blokas blokų grandinėje yra vadinamas „Pradžia“ (angl. Genesis).

M. Rennock'as, A. Cohn'as bei Jared Butcher'is savo darbe apie blokų grandinės technologiją [14] teigia, jog yra du tipai blokų grandinių tinklų – privatus arba viešas. Privatus tinklas skirtas tik tam tikrų naudotojų reikmėms, pavyzdžiui, bankams atlikti tranzakcijas. Tuo tarpu viešas tinklas yra atviras visiem, pavyzdžiui, „Bitcoin“, kuriame tranzakcijas gali atlikti bet kas. Tiek privatus, tiek viešas blokų grandinės tinklas pasižymi šiomis charakteristikomis:

- Realaus laiko įrašai – tinkle esantys nariai yra atnaujinami realiu laiku.
- Nekintantys įrašai – blokų grandinės technologija užtikrina, jog įrašai nebus pakeisti ar panaikinti, kadangi informacija nelaikoma vienoje vietoje, o apie ją žino visi tinklo nariai.



6 pav. Blokų grandinės veikimo schema

- Anonimiškumas – tiksliau įvarinant, tai pseudonimiškumas, kadangi visos tranzakcijos yra viešos ir žinomos, tačiau pačių naudotojų informacija yra užšifruota, taip suteikdama naudotojams anonimiškumą.

Darant išvadą galima teigti, jog blokų grandinės operuoja tam tikrais tranzakcijų blokais, kurie saugo ne tik bloko informaciją, tačiau ir prieš jį esančio bloko informaciją taip suformuojant nepertraukiamą grandinę, taigi mažas delsimas, anonimiškumas, decentralizuotas tinklas bei nekoreguojama informacija yra pagrindinės blokų grandinės charakteristikos.

1.3.2 Blokų grandinės technologija paremti DI įrenginių autentifikacijos metodai

Realaus laiko bei nekintatys įrašai – pagrindinės savybės, kurios tampa kertiniais argumentais pasirenkant naudoti blokų grandinės technologiją kuriant autentifikacijos mechanizmą. Tai patvirtina N. Malik savo darbe apie blokų grandinių principu paremtą apsaugotą autentifikacijos algoritmą [15]. Šiame darbe autorius teigia, jog technologija pasirinkta siekiant sumažinti autentifikacijos ir autorizacijos laiką bei išsaugant aukštą saugumo lygį, o darbui įgyvendinti buvo pasirinktas privatus blokų grandinių tinklas. Pačios technologijos principas yra paremtas tuo, jog mašinos gauna identifikacinį numberį iš sertifikato institucijos, kuris yra saugomas kartu su mašinos sertifikatu autentifikacijos blokų grandinėje. Vėliau, kai valstybinis numeris yra nuskaitomas kelyje, vykdoma mašinos autentifikacija ir identifikacija blokų grandinėje. Darbe atlikta našumo analizė, pavyzdžiui, su penkiomis mašinomis sistemos delsimas buvo tik 4.5ms.

Dar vienas metodas, paremtas blokų grandinių technologija ir reikalaujantis mažesnių resursų, tai S. Huh'o, S. Cho ir S. Kim'o darbe aprašytas ir siūlomas sprendimas [16]. Šiame darbe autoriai naudoja blokų grandines kontroliuoti ir konfigūruoti daiktų interneto įrenginius, o autentifikacija įgyvendinta naudojant RSA raktus: viešasis raktas yra laikomas „Ethereum“ platformojoe, o privatus raktas saugomas įrenginiuose. Sukurtas sprendimas paremtas tuo, jog blokų grandinė yra naudojama kaip programos duomenų bazės ir vieni įrenginiai perduoda komandas į blokų grandinę, o kiti įrenginiai – skaito informaciją ir laukia atnaujinimo. Šiame prototipe sukurti trys funkcionalumai: matuoklio išmatuotų reikšmių sekimas, taisyklių kūrimo bei taisyklių skaitymo

funkcionalumai. Šie funkcionalumai naudojami įvairių prietaisų tokia tvarka: matuoklio išmatuotų reikšmių sekimas bei taisyklių skaitymas – įvairūs matuokliai paremti „Raspberry Pi“ vienos plokštės kompiuteriu, taisyklių kūrimas – išmaniuoju telefonu naudojantis sukurta platforma. Visa komunikacija paremta RSA privataus-viešo rakto infrastruktūra. Pasak šaltinio autorių, vieša Eteriumo blokų grandinė neparodė pakankamo greičio bei užėmė daug vietos, taigi autoriai svarstė perkelti sprendimą į privačią blokų grandinę.

Ž. Radzevičius savo darbe „Blokų grandinėmis grįstas galinių įrenginių autentifikavimo ūko kompiuterijoje metodas“ [17] taip pat pateikė saugų autentifikacijos metodo sprendimą, kuris paremtas „Hyperledger Fabric“ blokų grandinės platforma bei naudoja AMQP protokolą daiktų interneto įrenginio komunikacijai su blokų grandinės klientu ūkio sluoksnyje. Autoriaus sukurtame sprendime autentifikavimas laikomas įvykusiu, kai blokų grandinėje patikrinus tranzakcijas pagal įrenginio identifikatorių, sudarytą iš užšifruotos įrenginio MAC adreso, operatyviosios atminties bei CPU informacijos, yra grąžinamas teigiamas rezultatas. Nors autorius neišskyrė problematinių sprendimo vietų, tačiau išanalizavus pateiktą sprendimą buvo pastebėta, jog nėra užtikrinama perduodamos informacijos sauga perduodant informaciją tarp įrenginio ir blokų grandinės kliento, kadangi informacija vyksta nesaugiu AMQP protokolu, bei žinučių apdorojimo trukmė yra ganėtinai ilga, pavyzdžiui, autentifikacija vyksta virš sekundės, o registracija net virš 3 sekundžių.

Visi ištirti darbai bei jų prototipai blokų grandinės technologiją naudoja informacijos atpažinimui ir identifikavimui, tačiau implementacija skiriasi: vienu atveju informacijos yra ieškoma tarp esamų blokų, o kitu – blokai yra sukuriami ir kiti nariai tinkle laukia naujo bloko su informacija. Taipogi greitis bei saugus komunikacijos kanalas šiuose darbuose išskirti kaip pagrindiniai neapgalvoti komponentai, kurie yra privalomi saugiam autentifikavimo metodui.

1.3.3 Blokų grandinių platformos

Ankstesnio skyriaus 1.3.2 šaltiniuose buvo išskiriamos įvairios blokų grandinių platformos. Tokių platformų, kurios savaip naudoja blokų grandinės technologiją, yra kelios, tačiau vienos populiariausių, pasak S. Wang'o, Y. Yuan'o ir kitų autorių [19], šiuo metu yra „Ethereum“ ir „Hyperledger Fabric“. Šiai nuomonei pritaria ir N. Teslya bei I. Ryabchikov'as [18], tačiau būtina atlikti analizę bei palyginti ir kitas populiarias platformas, siekiant išsirinkti reikalavimus atitinkančią:

- „Bitcoin Network“ – decentralizuotas tinklas, kuris paremtas *P2P* (angl. Peer-to-Peer) technologija ir privačiojo rakto kriptografija. Šis tinklas yra sudarytas iš „Bitcoin Core“ mazgų, kurie yra nemokami ir atviro kodo, o jų dauguma ir sudaro „Bitcoin Network“ tinklą. „Bitcoin Core“ dar vadinamas „Bitcoin“ klientu.

- „Ethereum“ – viena populiariausių ir lengviausiai implementuojamų platformų, siūlanti išmaniųjų kontraktų rašymą naudojantis Solidity [25] kalba. Pagrindinės charakteristikos – naudoja PoW(angl. Proof of Work) ir PoS(angl. Proof of State) hibridinį mechanizmą, palaiko Dapp(angl. Decentralized Application)

- „Hyperledger Sawtooth“ – tai platforma, skirta privačiam blokų grandinės tinklui. Sulaukė daug susidomėjimo po to, kai IBM paskelbė, jog naudoja šią platformą. Pagrindinės charakteristikos – išskiria vykdymą ir tranzakcijų validavimą į dvi atskiras fazes kas leidžia dauginti procesų vykdymą, o tranzakcijas rašyti su bet kuria programavimo kalba. Siūlo

	Ethereum	Hyperledger Sawtooth	Hyperledger Fabric	Bitcoin Network
Aprašymas	Globali ir decentralizuota platforma	Lanksti modulinė architektūra, atskirianti pagrindinę sistemą nuo programėlių srities	Leidimais grįsta blokų grandinės infrastruktūra, teikianti modulinę architektūrą	Mokėjimų blokų grandinės platforma
Consensuso algoritmai	PoW, PoS	PBFT, PoET	PBFT	PoW
Išmanieji kontraktai	Taip	Taip	Taip	Ne
Valiuta	Ether(ETH)	Nėra	Nėra	Bitcoin(BTC)

3 lentelė. Blokų grandinių platformų palyginimas

įvairių konsensuso algoritmų, tokių kaip PBFT(angl. Practical Byzantine Fault Tolerance) – praktinis gedimų toleravimas, arba modernesnį PoET(angl. Proof of Elapsed Time) – praėjusio laiko įrodymo metodas. Be to – tranzakcijų procesoriai taip pat valdo lokalų kiekvieno blokų grandinių dalyvio būseną, taigi be kreipimosi į blokų grandinių tranzakcijų sąrašą galima greitai pasiekti paskutinę būsenos informaciją.

- „Hyperledger Fabric“ – leidimais grįsta blokų grandinės infrastruktūra, teikianti modulinę architektūrą, dėl to ji yra lengvai konfigūruojama pagal reikiamas panaudos atvejus. Dažniausiai implementuojama kartu su „Kafka“ žinučių brokeriu.

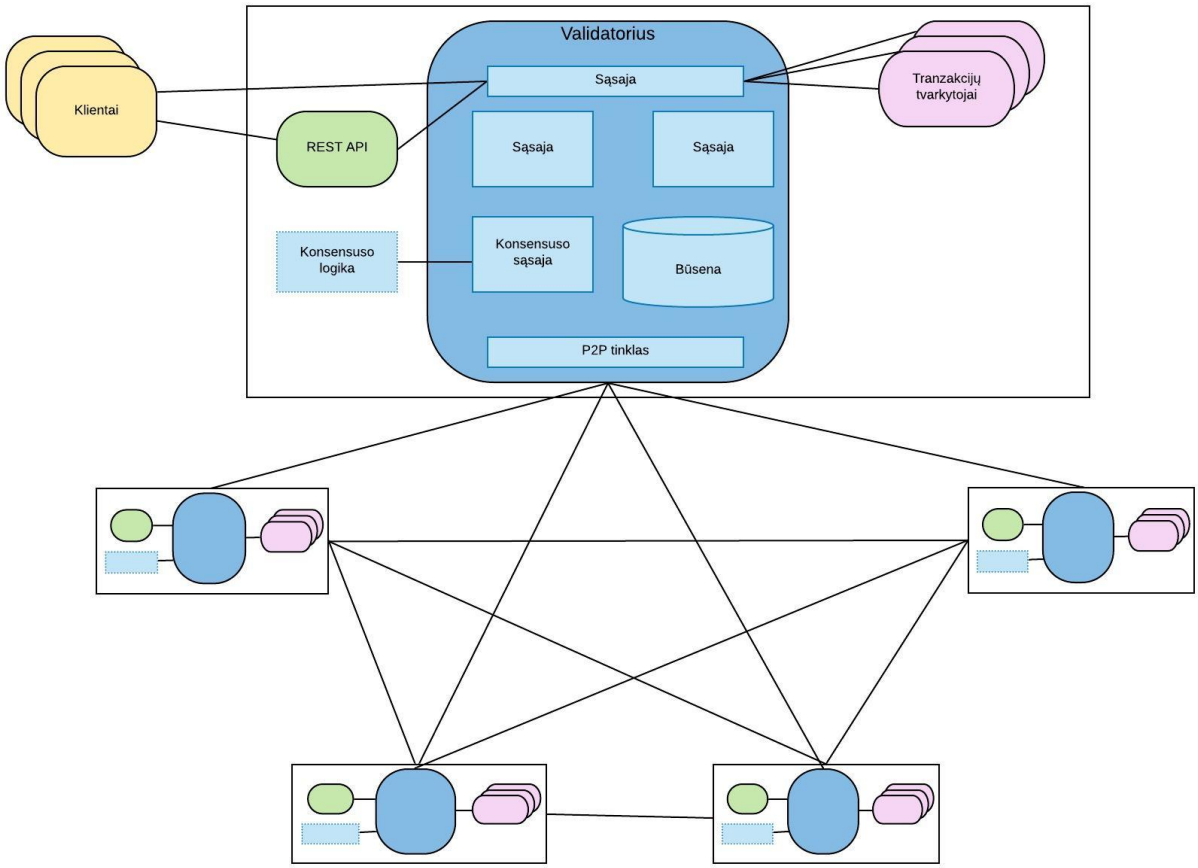
Iš šių išvardintų blokų grandinių platformų „Hyperledger Sawtooth“ yra labiausiai pritaikyta pramoniniam blokų grandinių funkcionalumui išpildyti, siekiant užtikrinti paskirstytą duomenų saugojimą bei integralumo išlaikymą. Šios platformos projektavimo filosofija siekia išlaikyti duomenis paskirstytus kartu užtikrinant išmaniųjų kontraktų saugumą, ypač pramoniniam panaudojimui. „Sawtooth“ unikalumas, išskiriantis ją tarp konkurentų, yra pagrindinės blokų grandinės logikos atskyrimas nuo verslo logikos bei reikalavimų, taip leidžiant programuojant funkcinius reikalavimus net nežinant apie blokų grandinės funkcionalumo ypatumus, susitelkiant į tranzakcijų tvarkytojų logiką, kuriuos galima laikyti lyg atskiromis programėlėmis, ir naudojantis blokų grandine lyg paprasta duomenų baze.

„Hyperledger Sawtooth“ projektavimo architektūra pavaizduota paveikslėlyje 7, kuriame detaliai pavaizduotas mazgas:

- tranzakcijų tvarkytojai – tai PnP(angl. Plug and Play) principu veikiantys tvarkytojai, kuriuos galima rašyti bet kuria programavimo kalba ir jie atsakingi už jiems skirtų tranzakcijų apdorojimą, pasirašymą bei validatoriaus būsenos atnaujinimą.

- REST API – REST sąsaja, suteikianti galimybę pasiekti skirtingus mazgus klientinėms dalims

- validatorius – pagrindinis „Sawtooth“ componentas, apjungiantis visus mazgus ir sudarytas iš šių komponentų:



7 pav. „Hyperledger Sawtooth“ architektūra

- sąsaja – programinė įranga, skirta klientų bei tranzakcijų tvarkytojų komunikaciniai su validatoriumi
- blokų tvarkyklė – atsakinga už blokų tvarkymą, validavimą bei paskelbimą
- tranzakcijų apdorotojas – atsakingas už tranzakcijų pasirašymą bei gautų tranzakcijų paskirstymą pagal tranzakcijos šeimą
- konsensuso sąsaja – dinaminis konsensusas yra dar viena iš „Sawtooth“ ypatybių, o ši sąsaja atsakinga už konsensuso prijungimą į bloko pasirašymo fazę
- būsena – tai visų tranzakcijų šeimų atnaujinama viena būsena, kuri išsaugoma *Merkle-Radix* medyje. Būsena skirstoma pagal tranzakcijų šeimas, taip apsaugant skirtingų tranzakcijų tvarkytojų būsenas
- P2P(angl. Peer-to-Peer) tinklas – sąsaja užtikrinti pasiekiamumą tarp kiekvieno tinklo mazgo bei ištransliuoti ar gauti žinutes

Detaliau būtina išskirti būsenos valdymą, kuris įgyvendinamas *Merkle-Radix* medžio principu. Būsenos medis yra Merkle, nes ši duomenų struktūra saugo nuoseklias mazgo maišos reikšmes nuo medžio lapų iki šaknų vos tik atlikus bet kokį pakeitimą medžio struktūroje, taip apskaičiuojant medžio versiją, kuri būtų šakninė maišos reikšmė. Šią reikšmę siunčiant tranzakcijos antraštėje – užtikrinama, kad globali būsena yra vienoda visuose mazguose. Jeigu šakninės maišos reikšmė

neatitinka mazgo validatoriaus apskaičiuotai reikšmei – tranzakcija yra laikoma nepavykusi ir atmetama tinklo dalyvių. Medis taip pat yra ir adresuotas Radix medis, nes adresai unikaliai nurodo kelią iki lapų medyje, kuriuose informacija yra saugoma. Adresas yra šešioliktainė 70 simbolių eilutė, kurioje pirmi 3 baitai nurodo vardo sritį, o kiti 32 baitai – yra užkoduotas kelias iki reikšmės.

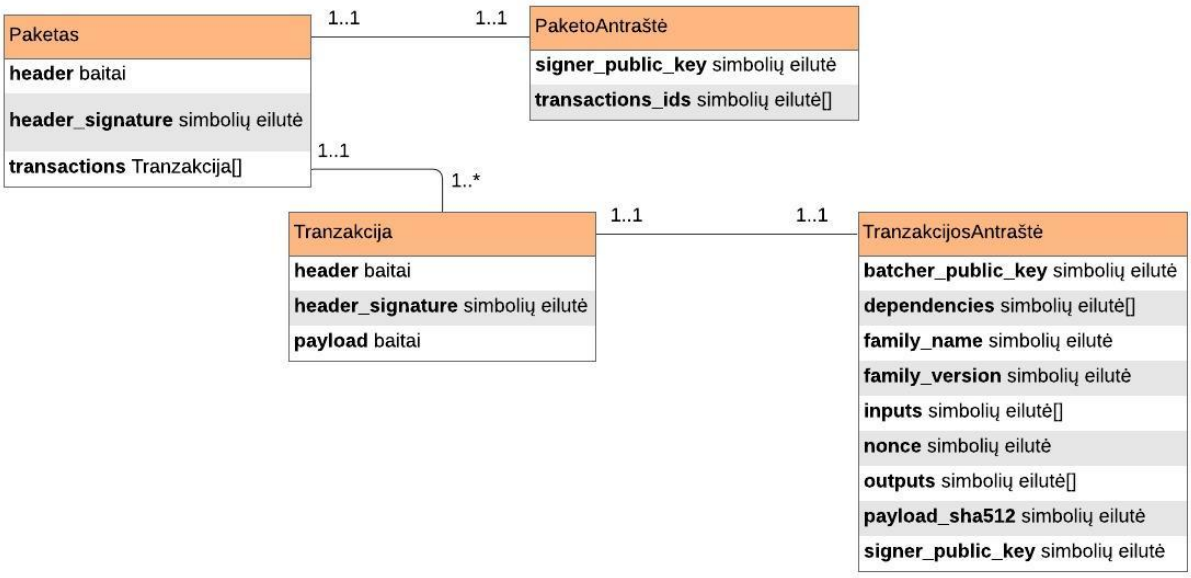
Būsenos atnaujinimai yra atliekami paketais, t.y. mazgo klientai su tranzakcijų tvarkytojais sukuria reikiamas tranzakcijas ir jas sudeda į vieną paketą, kuris yra išsiunčiamas į tinklą patvirtinimui. Tai iliustruoja 8 paveikslėlis, kuriame pavaizduota paketo sandara. Paketas yra sudarytas iš:

- paketo antraštės (`header`) – ji saugo informaciją apie paketo tranzakcijų identifikatorius bei paketo sudarytojo viešą raktą
- paketo antraštės parašo (`header_signature`) – tai antraštės reikšmė pasirašyta su paketo sudarytojo privačiu raktu naudojant secp256k1 kreivę
- tranzakcijų sąrašo – sąrašas Tranzakcijos objektų sudarytų iš
  - antraštės (`header`) – ji saugo daugiausiai informacijos pradedant tranzakcijos formuotojo viešu raktu (`batcher_public_key`), kitų privalomai pirmiau įvykdomų tranzakcijų identifikatoriai (`dependencies`), tranzakcijų tvarkytojų šeimos pavadinimas (`family_name`), tranzakcijų tvarkytojų šeimos versija (`family_version`), būsenos adresai kuriuos tranzakcija gali skaityti (`inputs`), atsitiktinė reikšmė skirta užtikrinti parašo unikalumą (`nonce`), būsenos adresai į kuriuos tranzakcija gali rašyti (`outputs`), užšifruotas turinys su sha512 kripto funkcija (`payload_sha512`), tranzakcijos pasirašytojo viešas raktas (`signer_public_key`)
  - antraštės parašo (`header_signature`) – tranzakcijos turinys pasirašytas su tranzakcijos sudarytojo privačiu raktu naudojant secp256k1 kreivę
  - turinys (`payload`) – užšifruotas tranzakcijos šeimos specifinis turinys.

Įdomu yra tai, jog „Sawtooth“ nesiunčia į tinklą po vieną tranzakciją, bet siunčia paketais (angl. batches), kurie yra atominiai pokyčių vienetai, tai reiškia, jog visos tranzakcijos, atsiunčiamos pakete, reikš vieną pasikeitimą sistemoje taip užtikrinant atsiųstų duomenų vientisumą ir nebus atliekamas dalinis duomenų pakeitimas. Jeigu bent viena iš paketo tranzakcijų nebuvo patvirtinta – nė viena paketo tranzakcija nebus pridėta prie tinklo tranzakcijų ir globali būsena nebus pakeista. Šitai suteikia unikalų funkcionalumą – skirtingų tranzakcijų tvarkyklių šeimų tranzakcijas nusiųsti vienu paketu ir taip užtikrinant atominį duomenų pasikeitimą

Renkantis konsensuso algoritmą galima rinktis iš 2 pagrindinių metodų:

- PBFT – balsavimo principu paremtas metodas, kurio metu reikalaujame 2/3 tinklo narių patvirtinimo, o už balsavimą yra atsakingas pagrindinis mazgas
- PoET – vienas iš sąžiningiausių konsensuso algoritmų, kuris paremtas loterijos principu, t.y. kiekvienas validatorius prašo atsitiktinio laukimo laiko ir tam laikui praėjus gaus prieigą paskelbti naują bloką.



8 pav. „Hyperledger Sawtooth“ paketo struktūra

1.4 Analizės išvados

Analizė parodė, jog pagrindinė daiktų interneto saugumumo problema įvardinama kaip išmaniųjų tinklo įrenginių nesaugus bendravimas, centralizuotas serveris bei autentifikavimas, o dabartiniai metodai nors ir siekia užtikrinti apsaugą nuo neatpažintų narių ir įvairių atakų, tačiau dažnai neapgalvoja resursais apribotų įrenginių įtraukimą į tinklą bei atlieka tik vieno faktoriaus autentifikaciją. Taip pat išsiaiškinta, jog daiktų interneto architektūra skirstoma į tris sluoksnius, kuriuose operuoja skirtingi protokolai. Programos sluoksnyje plačiausiai paplitę yra AMQP bei MQTT protokolai, o renkantis žinučių brokerį didžiausia konkurencija vyksta tarp „RabbitMQ“ bei „ActiveMQ“ atviro kodo brokerių.

Ištyrus blokų grandines galima teigti, jog šios technologijos panaudojimas padėtų decentralizuoti pagrindinį autentifikacijos šaltinį, autentifikaciją perkelti į decentralizuotą tinklą, kuriame būtų galima įrenginio informaciją talpinti blokuose taip atsiribojant nuo didelės informacijos talpinimo įrenginio viduje, išsprendžiant esamų metodų problemas, ir kartu užtikrinti informacijos vientisumą bei konfidencialumą. Peržiūrėjus blokų grandinėmis paremtus autentifikacijos metodus padaryta išvada, jog autentifikacijos greitaveika labai priklauso ne tik nuo blokų grandinių platformos, tačiau ir nuo to ar privačiame, ar viešame blokų grandinių tinkle autentifikacija atliekama.

Apibendrinant atliktą analizę buvo pastebėta, jog esami daiktų interneto sprendimai nėra pakankamai saugūs, kadangi autentifikaciją atlieka tik vienu žinomu požymiu, o informacija dažniausiai perduodama per internetą ir operuojama viešomis blokų grandinių platformomis bei neapsaugojus komunikacijos kanalo.

Uždaviniai, keliami realizacijos etapui yra šie:

- suprojektuoti siūlomą autentifikacijos platformą – ši platforma privalo išspręsti aptartas autentifikacijos problemas: nesaugus išmaniųjų įrenginių komunikavimas, nepakankamai saugi autentifikacija bei centrinis serveris, sukuriantis pavienio pažaidos taško (angl. SPOF – single point of failure) problemą;

- sukurti pasiūlyto autentifikavimo metodo prototipą – tai įrodytų, jog teorinį modelį tikrai galima įgyvendinti bei padėtų atlikti kiekybinį tyrimą;
- pateikti sukurto sprendimo tyrimo įvertinimą – išskiriant kokybines bei kiekybines eksperimentinio tyrimo analizes.

2 Daiktų interneto įrenginių autentifikavimo modelis panaudojant blokų grandinės technologiją

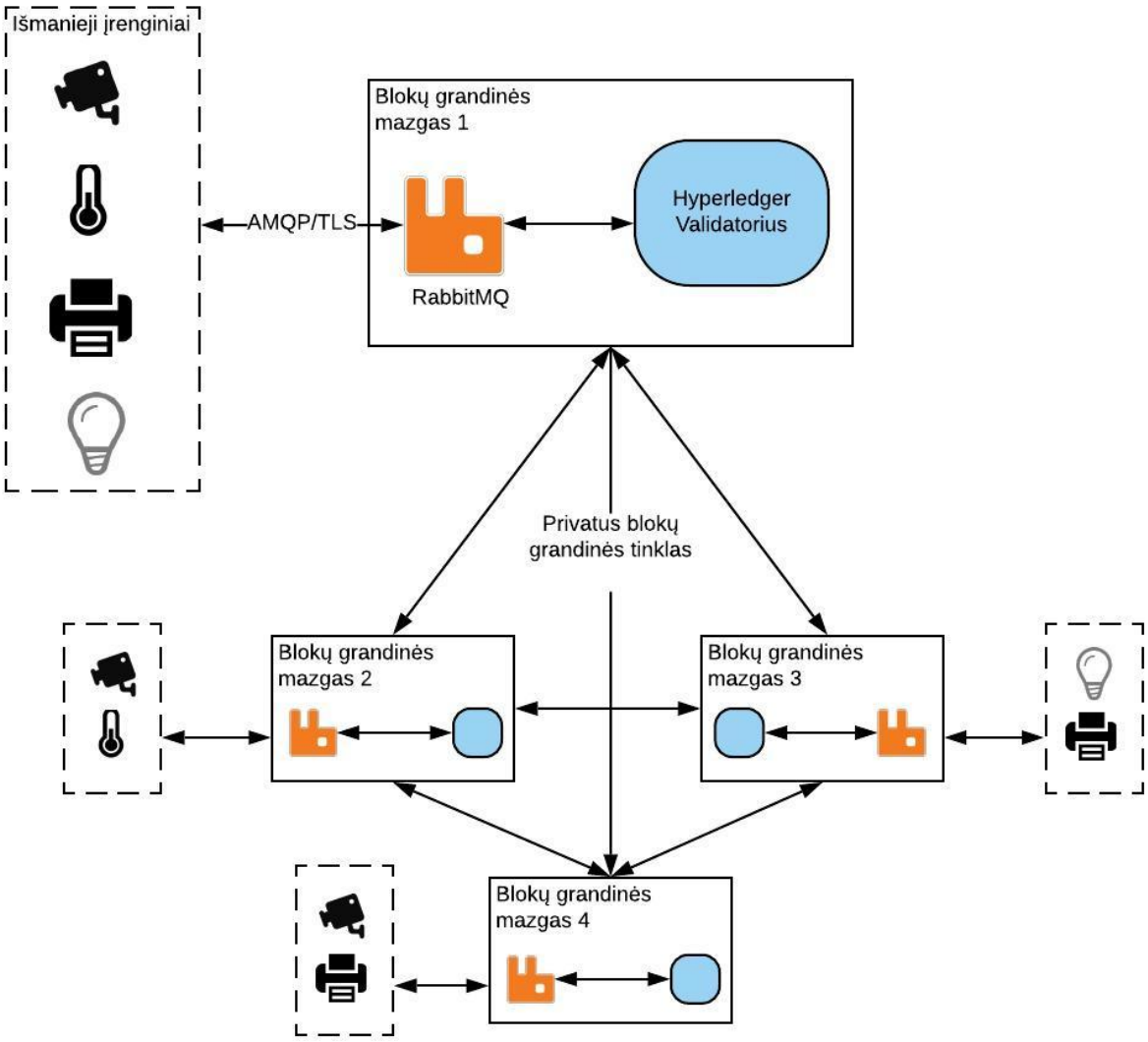
Ištyrus eksperimentinius daiktų interneto autentifikacijos bei blokų grandinių panaudojimo daiktų internete metodus buvo nuspręsta sukurti blokų grandinėmis grįstą daiktų interneto įrenginių autentifikacijos mechanizmą, atitinkantį kasdien kylančius DI saugumo reikalavimus, kuris:

- būtų paremtas F. Santos'o ir N. Vun'o darbu [27] apie daiktų interneto įrenginių autentifikavimą, kadangi sprendimas operuoja Wi-Fi tinklu, kuris yra vienas praktiškiausių panaudojimų;
- būtų paremtas M. Nisha darbu [15] apie automobilių autentifikaciją privačiame blokų grandinių tinkle remiantis išduotais sertifikatais. Šis sprendimas išspręstų aukščiau pateikto darbo decentralizacijos problemą bei parodė galimą sertifikatų panaudojimą;
- naudotų „Hyperledger Sawtooth“ privačią blokų grandinės platformą, kadangi ištirti metodai teigė, jog vieša „Ethereum“ platforma neparodė reikiamo greičio bei vykdymas ir validavimas vykdomas viena faze;
- įrenginio bei blokų grandinės tinklo klientas bendrautų AMQPS protokolu per „RabbitMQ“ žinučių brokerį saugiu TLS kanalu.

Atlikus analizės dalį ir išsikėlus reikalavimus implementacijos daliai, pirmiausia pradėta nuo realizacijos aukšto lygio diagramos (angl. HLD – high level diagram), pavaizduotos paveikslėlyje 9.

Šioje principinėje aukšto lygio diagramoje atskleidžiamos pagrindinės sprendimo ypatybės: dviejų faktorių autentifikacija ir saugus komunikacijos kanalas. Įvairūs išmanieji įrenginiai galės komunikuoti su priskirtu blokų grandinės mazgu. Blokų grandinės mazgas (angl. blockchain node) bus priskirtas privačiam blokų grandinės tinklui ir bendraus su kitais mazgais, taip dalindamiesi globalia būsena bei atnaujinimais apie išmaniųjų įrenginių informaciją bei statusą. Mazgas bus sudarytas iš dviejų komponentų, kurie atliks autentifikavimo modulių funkcijas ir įgyvendins dvejų faktorių autentifikaciją:

1. pirmasis modulis – „RabbitMQ“ serveris tikrintų turimus išmaniojo įrenginio kliento prisijungimus prie mazgo pagal jam išduotą sertifikatą. Su šituo serveriu išmanieji įrenginiai komunikuos apsaugotu kanalu
2. antrasis modulis – „Hyperledger Sawtooth“ validatorius, kuris tikrintų išmaniojo įrenginio identitetą pagal jo atpažinimo ID su tinkle patvirtinta globalia būsena.



9 pav. Realizacijos aukšto lygio diagrama

Projektavimo fazėje buvo sudėliota panaudos atvejų diagrama (žr. pav. 10), kuri padeda aprašyti ką projektuojama sistema gali atlikti, kokie sistemos veikėjai egzistuoja ir jų galimybę atlikti tam tikras funkcijas. Diagramos aprašas pateiktas 4 lentelėje.

PA „Daiktų interneto įrenginio užklausos apdorojimas“		
Tikslas: Autentifikuoti įrenginių užklausas ir jas apdoroti		
Aprašymas: įrenginys siunčia užklausą į serverį ir laukia atsakymo, o tuo metu AMQP serveris bei blokų grandinės mazgo klientas autentifikuoja įrenginį bei apdoroja užklausą		
Prieš sąlygą:		Įrenginys turi būti priregistruotas blokų grandinėje bei įtrauktas į tinką
Sužadinimo sąlyga:		Įrenginio užklausa
Aktoriai:		Įrenginys, AMQP komunikacijos serveris, blokų grandinės mazgo klientas
Susiję PA	**Išplečiantys PA**	Atlikti antrinę įrenginio autentifikaciją, atlikti pirminę įrenginio autentifikaciją, autentifikuoti įrenginį
	Apimami PA	Patikrinti įrenginio informaciją blokų grandinės tinkle, apdoroti gautus duomenis
	Specializuoti PA	Apdoroti gautą užklausą
Pagrindinis įvykių srautas		**Sistemos reakcija**
1. Įrenginys išsiunčia užklausą		1.1. Sistema atlieka pirminę autentifikaciją
		1.2. Sistema atlieka antrinę autentifikaciją
		1.3. Sistema apdoroja užklausą
		1.4. Sistema išsiunčia atsakymą
		1.5. Sistema sukuria žurnalizavimo įrašą
2. Įrenginys apdoroja gautus duomenis		
Po sąlygos:		Atsiranda žurnalizavimo įrašas, duomenys pasikeičia priklausomai nuo siųstos įrenginio užklausos (stebėsenos įrašas, komandos įrašas, kt.)
Alternatyvūs scenarijai		
1. Įrenginys nusiuntė neteisingus AMQP autentifikacijos duomenis		1.1. AMQP serveris grąžino nesėkmingos autentifikacijos atsakymą
2. Įrenginys nebuvo registruotas blokų grandinėje		2.1. Blokų grandinės mazgo klientas grąžino nesėkmingos autentifikacijos atsakymą
3. Įrenginys atsiuntė nevalidžią užklausą		3.1. Blokų grandinės mazgo klientas grąžino klaidos atsakymą

4 lentelė. Daiktų interneto įrenginio užklausos apdorojimo panaudos atvejo diagramos aprašymas

10 pav. Daiktų interneto įrenginio užklausos apdorojimo panaudos atvejo UML diagrama

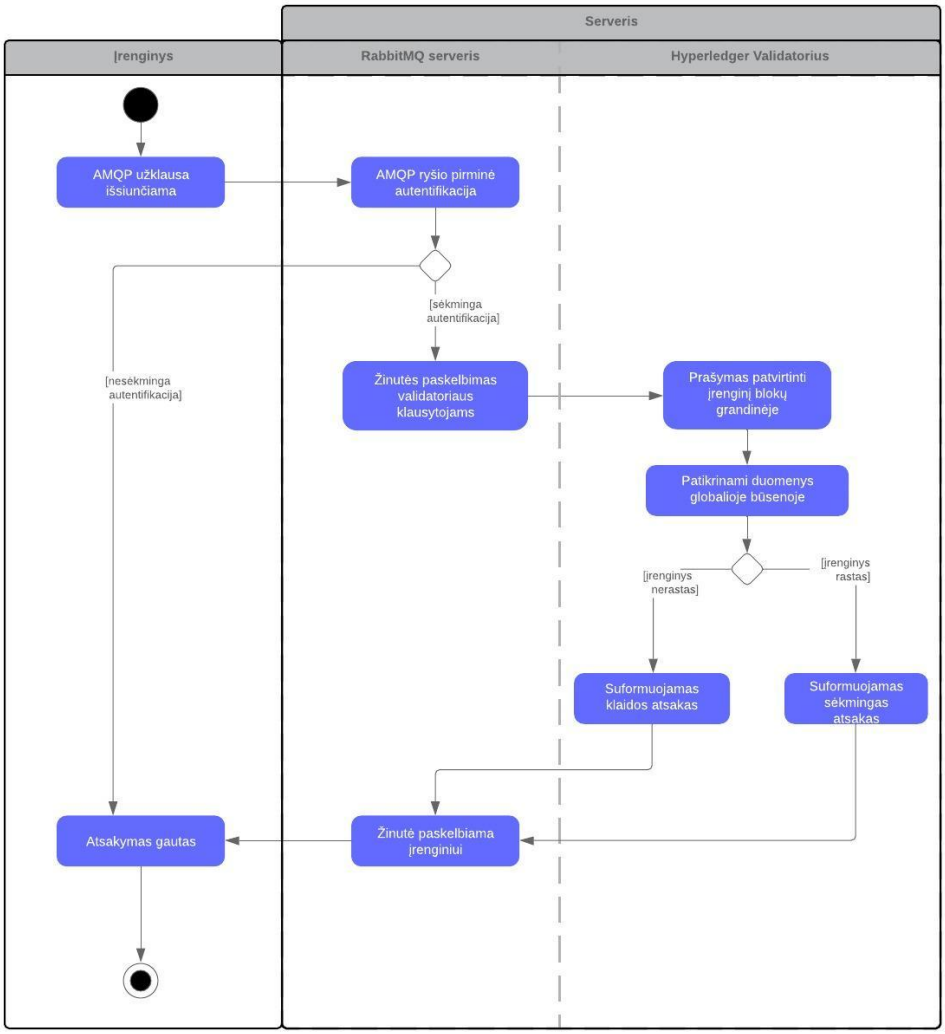
2.1 Daiktų interneto autentifikavimo modelis

Projektuojamas autentifikavimo modelis susideda iš dvejų lygių patvirtinimo:

1. „RabbitMQ“ serverio autentifikacija – patikrinama jungties informacija tarp serverio ir kliento
2. Blokų grandinės autentifikacija – patikrinama įrenginio informacija blokų grandinėje

Žemiau pavaizduotoje autentifikavimo veiklos diagramoje (žr. pav. 11) yra pavaizduojama principinė autentifikavimo idėja parodant konkrečius veiksmus tiek iš fizinio įrenginio, tiek iš fizinio serverio servisų („RabbitMQ“ serviso bei „Hyperledger Sawtooth“ validatoriaus). Pirmoji

veikla yra įrenginio siunčiama autentifikavimo užklausa, kuri AMQPS protokolu perduodama į „RabbitMQ“ serverio eilę, tačiau prieš tai yra atliekamas pirmasis autentifikacijos žingsnis – inicijuojama autentifikacija su „RabbitMQ“ serveriu saugiu kanalu. Jeigu sertifikato naudotojas yra registruotas „RabbitMQ“ serveryje – ši žinutė yra paskelbiama klausytojui, kuris įjungtas blokų grandinės mazge. Šis klausytojas atsakingas už žinučių surinkimą ir užklausos suformavimą mazgo validatoriui per REST servisą. Validatorius, patikrinęs duomenis tranzakcijose ir globalioje blokų grandinės būsenoje, grąžina atsakymą mazgo klientui, kuris paskelbia naują atsako žinutę į „RabbitMQ“ serverį apie gautą atsakymą.



11 pav. Autentifikavimo veiklos UML diagrama

2.1.1 Įrenginio identifikacija

Prieš detaliau projektuojant autentifikaciją būtina apibrėžti globalų įrenginio saugų identifikacinį numerį, kuris bus pagrindinė įrenginio žymė tiek pirminės, tiek antrinės autentifikacijos metu, kadangi skirtingi identifikatoriai gali reikšti sistemos vientisumo pažeidimą bei sudėtingesnį įrenginių valdymą.

Įrenginio identifikatorius bus HMAC-SHA256 maišos funkcija užšifruoti du duomenys: unikalus įrenginio ID ir paslaptis.

Unikalus įrenginio ID tai sisteminis įrenginio identifikatorius, kuris sugeneruojamas ir priskiriamas sistemos instaliacijos metu. Skirtingose operacinėse sistemose šios reikšmės randamos:

- Linux – naudojama reikšmė iš `/var/lib/dbus/machine-id`
- Windows – naudojama `MachineGuid` reikšmė iš registro `HKEY_LOCAL_MACHINE \ SOFTWARE\Microsoft\Cryptography`
- OS X – `IOPlatformUUID` sisteminė reikšmė.

HMAC-SHA256 algoritmas kartu su įrenginio ID pasirinktas remiantis Linux aplinkų [33] generuojama saugia maišos reikšme, kuri naudoja HMAC-SHA256 algoritmą. Blokų grandinės klientų grupės paslaptis bus naudojama kaip maišos funkcijos slaptas raktas.

HMAC-SHA256 algoritmas gali būti išreikštas kaip:

$$HMAC(K, M) = H((K \oplus opad) \parallel H((K \oplus ipad) \parallel M))$$

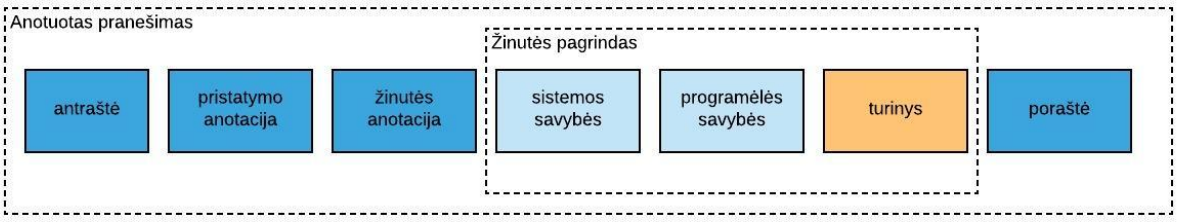
Pavaizduotoje formulėje:

- H – SHA256 funkcija
- K – slaptas raktas
- M – žinutė, kuri bus įvestis HMAC funkcijai, tai unikalus įrenginio ID
- $\parallel$ – reiškia apjungimą
- $\oplus$ – atspindi XOR funkciją

Šios funkcijos išvestis – tai globalus įrenginio identifikacinis numeris, kuris bus naudojamas tiek autentifikacijos, tiek registracijos metu.

2.1.2 Serverio autentifikacija

AMQP protokolas pasirinktas duomenų perdavimui iš daiktų interneto įrenginio į jam priskirtą serverį, tačiau būtina išsiaiškinti kaip informacija sudėliojama ir kokia žinutės struktūra keliauja duomenys šiuo protokolu.



12 pav. AMQP žinučių struktūra

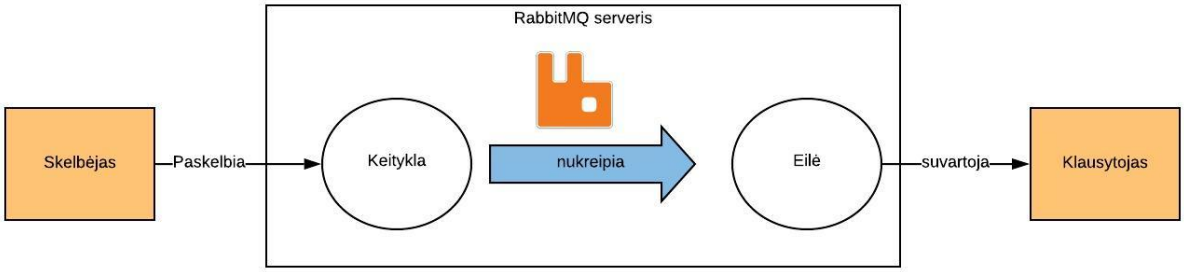
Aukščiau pavaizduotoje struktūroje (žr. pav. 12) matomos pagrindinės AMQP protokolu keliaujančios žinutės dalys: anotuotas pranešimas ir žinutės pagrindas. Žinutės pagrindas tai pagrindinė informacija, suformuota siuntėjo ir nekintanti keliaujant tarp stotelių ar serverių, tuo tarpu

anotuoto pranešimo dalys gali kisti priklausomai nuo sudėliotos tinklo struktūros. Nekintantis žinutės pagrindas leidžia siuntėjams ir gavėjams pasirašyti žinutę išsiunčiant bei patikrinti parašą ją gavus, o anotuota žinutės dalis papildomą ar pakeičiama žinučių brokerio stotelėse. Smulkesnės žinutės dalys[35]:

- antraštė – saugo penkis pagrindinius žinutės atributus: patvarumo reikalavimus, žinutės pirmumą, TTL (angl. time to live), pirmojo gavėjo požymį bei skaičių, kiek kartų nesėkmingai žinutė buvo pristatyta
- pristatymo anotacija – neprivaloma nestandartinė antraštės reikšmė, skirta perteikti reikšmes nuo siuntėjo gavėjui
- žinutės anotacija – neprivaloma nestandartinė antraštės reikšmė, naudojama nustatyti žinutės ypatybes, kurios skirtos infrastruktūrai ir turi būti išplatintos kiekviename skirstymo etape
- savybės – laiko įvairias neprivalomas fiksuotas reikšmes apie žinutę: žinutės identifikatorius, siuntėjo identifikatorius, gavėjas, tema, nuoroda kam atsakyti, programos koreliacijos identifikatorius, turinio tipas, turinio kodavimas, absoliuti žinutės galiojimo data, sukūrimo data, grupės identifikatorius, grupės žinutės eilės numeris, grupės atsakymo adreso identifikatorius
- programėlės savybės – papildomos programėlės savybės, nesusijusios su turiniu
- turinys – žinutės turinys
- poraštė – neprivaloma reikšmė, skirta detalėms apie žinutę, kurios nuskaitomos kai visa inutė jau buvo sukonstruota ar nuskaityta. Dažniausiai – žinutės maišos reikšmės ar parašai.

Iš šios struktūros reiktų išskirti savybės komponentą, kadangi jis laiko siuntėjo identifikatorių, taigi siuntėjui paskelbus žinutę – gavėjo klientai gali pasiekti siuntėjo identifikatorių.

Prieš pasirenkant saugų autentifikavimo metodą – pirmiausia būtina suprasti kaip „RabbitMQ“ serveris priima bei išskirsto žinutes, šis modelis pavaizduotas paveikslėlyje 13.



13 pav. „RabbitMQ“ žinučių nukreipimas

Žinutei patekus į „RabbitMQ“ serverį ir sėkmingai autentifikavus žinutės siuntėją – serveris ją nukreipia į eiles, kurias prenumeruoja klausytojai ir jų klausosi. Būtina paminėti, jog yra keli skirtingi skirstymo algoritmai, kurie dažniausiai priklauso nuo funkcinių reikalavimų žinutei ar net visai aplikacijai: tiesioginis, vėduoklės, pagal temas ir pagal antraštes. Pirmasis ir numatytasis algoritmas serveryje yra tiesioginis, kurio skirstymo logika yra paprasta – nukreipia žinutes

į eiles, kurių klausymosi raktas (angl. routing key) sutampa su siuntimo raktu, t.y. jeigu žinutė buvo atsiųsta su raktu *R* – ji bus nukreipta į visas eiles, kurios skirtos *R* nukreipimo raktui. Vėduoklės principas visiškai kitoks – pasirinkus šį algoritmą keitykla nukreips visas žinutes į visas eiles ir ignoruos nukreipimo raktą. Tuo tarpu temų algoritmas yra gan panašus į tiesioginį, tačiau leidžia nukreipimo raktą nustatyti struktūriškai, t.y. jeigu žinutė atsiųsta su nukreipimo raktu *messages.private*, tai ją gaus eilės, klausančios nukreipimo raktų *messages.**, *#* bei *messages.private*. Apibrėžiant temos algoritmo nukreipimo raktus žvaigždutės simbolis atitinka bet kokį žodį, o grotažymė atitinka nulį ar daugiau žodžių, perskirtų tašku. Paskutinis algoritmas – pagal antraštes, taigi pasirinkus šį algoritmą nukreipimo rakto reikšmė bus ignoruojamas, o nukreipimas vyks pagal antraštėse nurodytas reikšmes. Verta paminėti, jog antraštės, kurių raktas prasideda *x-* bus ignoruojamos.

Išsiaiškinus žinučių skirstymą matoma, jog autentifikavimas atliekamas paskelbus žinutę ir jai atkeliavus į serverio keityklą. Jau išsiaiškinta analizės skyriuje 1.2.1, jog „RabbitMQ“ palaiko *AMQPLAIN*, *PLAIN* bei x.509 sertifikatais paremtą autentifikaciją, o pagrindinė dilema renkantis tarp šių mechanizmų – autentifikuoti klientą pagal vardą ir slaptažodį ar pagal išduotą sertifikatą bei jame užkoduotą vardą.

Daugumoje net ir šiais laikais kuriamų sistemų autentifikaciją atlieka reikalaudamos prisijungimo vardo bei slaptažodžio, kuris būna kliento sugalvotas ar sistemos priskirta slapta kombinacija žodžių ir skaičių, kuri žinoma tik kliento sistemos. Viena pagrindinių problemų, išskirtų Nilesh Lal'o ir kitų autorių straipsnyje apie autentifikacijos metodus [31], tai galimybė per palyginamai trumpą laiką atspėti slaptažodį ar perimti jį klausantis tinklo. Šiuos trūkumus gali išspręsti tik saugus ryšys, plačiau žinomas kaip TLS, bei naudotojo autentifikavimas pagal tik jam būdingą identitetą. Šis protokolas apsaugo duomenų perdavimą tarp mazgų kriptografiškai ir užtikrina saugią autentifikaciją, išlaikant pagrindines saugumo sąlygas, kaip aprašyta TLS 1.3 versijos apraše [32]: autentifikaciją, konfidencialumą ir vientisumą. Šis protokolas susideda iš 2 komponentų: inicializacijos (angl. handshake) protokolo, užmezgančio sesiją ir autentifikuojančio abi puses, ir įrašo protokolo, kuris užtikrina saugų kanalą atliekant tolimesnę komunikaciją. Šio darbo tematikoje aktualiausias yra inicializacijos protokolas, kuris užtikrina, jog sesiją leis inicializuoti tik autentifikuotiems įrenginiams atsižvelgiant į saugumo parametrus. Autentifikaciją galima užtikrinti keliais palaikomais asimetriniais kriptografiniais metodais: RSA, ECDSA, EdDSA, arba simetriniu PSK.

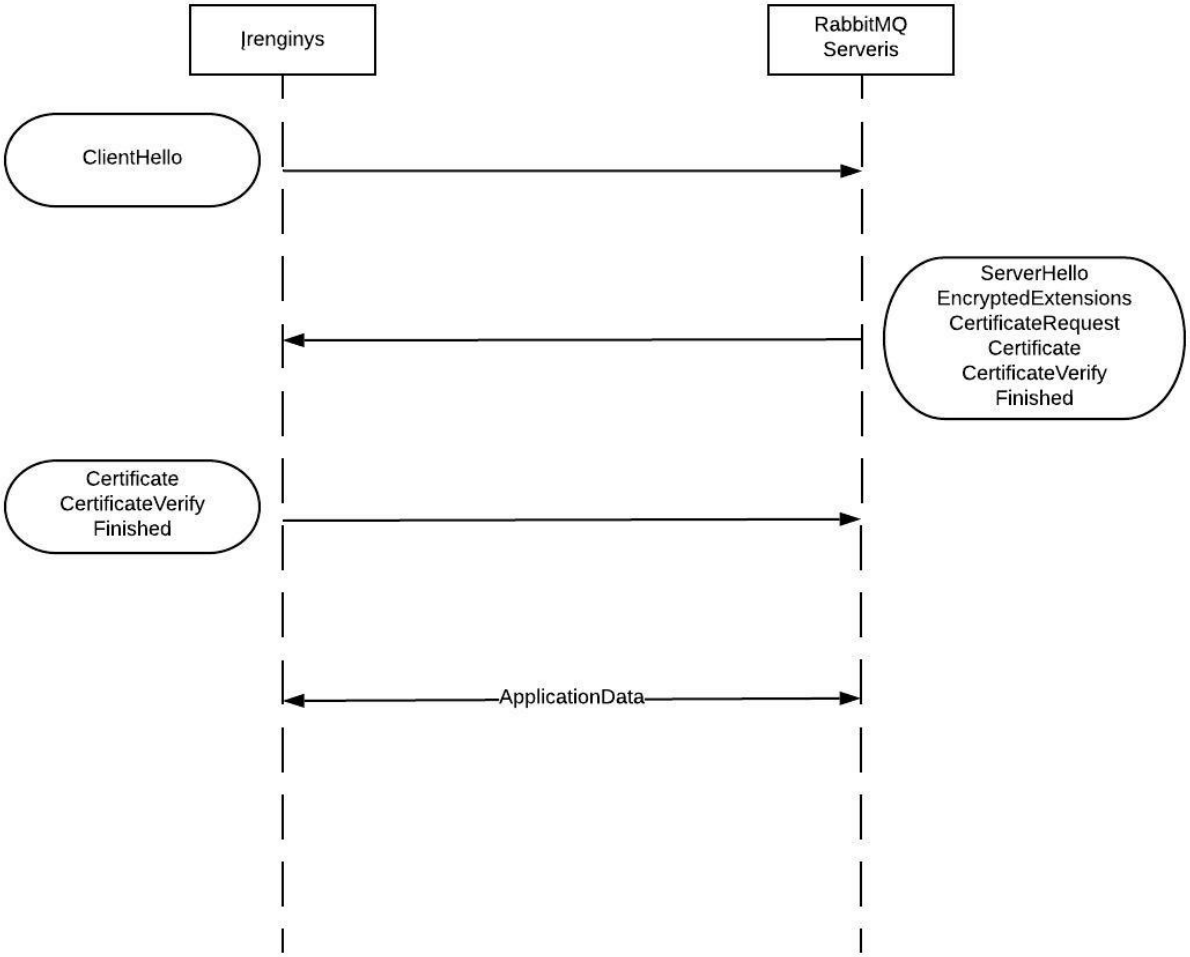
Pagrindinė inicializacijos protokolo schema pavaizduota paveikslėlyje 14:

1. Įrenginys kreipiasi į serverį siųsdamas *ClientHello* žinutę kartu pateikdamas savo siūlomas protokolo versijas, sąrašą simetrinių maišos porų (pavyzdžiui, Diffie-Hellman raktus)

2. Serveris apdoroja *ClientHello* žinutę ir nusprendžia kurie kriptografiniai parametrai bus naudojami komunikacijai, o tada atsako su *ServerHello* žinute, kuriame yra perduodami susitarti kriptografiniai parametrai. Vėliau serveris išsiunčia dar dvi žinutes: *EncryptedExtensions* žinutė perduoda *ClientHello* papildinius, kurie nėra reikalingi nustatant kriptografinius parametrus, o *CertificateRequest* žinutė perduoda norimus parametrus pasirinkto sertifikato, jeigu pasirinktas metodas reikalauja autentifikacijos

3. Galiausiai, klientas ir serveris apsikeičia autentifikacijos žinutėmis: *Certificate* – pasidalinama sertifikatais, *CertificateVerify* – parašas visos inicializacijos pasinaudojant savo privačiais raktais (kurių vieši raktai buvo pasidalinti). Galiausiai išsiunčiama *Finished* žinutė su

žinutės autentifikacijos kodu (MAC) taip patvirtindama siuntėjo tapatybę bei autentifikuodama inicializaciją ir ją užbaigdama.



14 pav. TLS 1.3 pilnas ryšio užmezgimas

Po inicializacijos *Finished* žinučių jau prasideda *ApplicationData* duomenų dalinimasis per jau saugų šifruotą kanalą.

Projekto pasirinktas AMQP žinučių brokeris „RabbitMQ“ serveris leidžia autentifikuoti sesijas TLS sertifikatais, todėl saugų kanalą bei autentifikaciją įmanoma įjungti įdiegus kelis papildomus modulius tuo pačiu patobulinant komunikacijos protokolą į saugesnį AMQPS (angl. AMQP over TLS/SSL encrypted). Panaudojus TLS autentifikacijos mechanizmą – „RabbitMQ“ serveris klientus autentifikuos x.509 sertifikatais ir komunikacijos kanalą šifruos aukščiau aprašytu TLS, o klientą identifikuos pagal sertifikate užkoduotą kliento vardą (angl. Common Name) ir praleis slaptažodžio tikrinimą.

2.1.3 Blokų grandinės autentifikacija

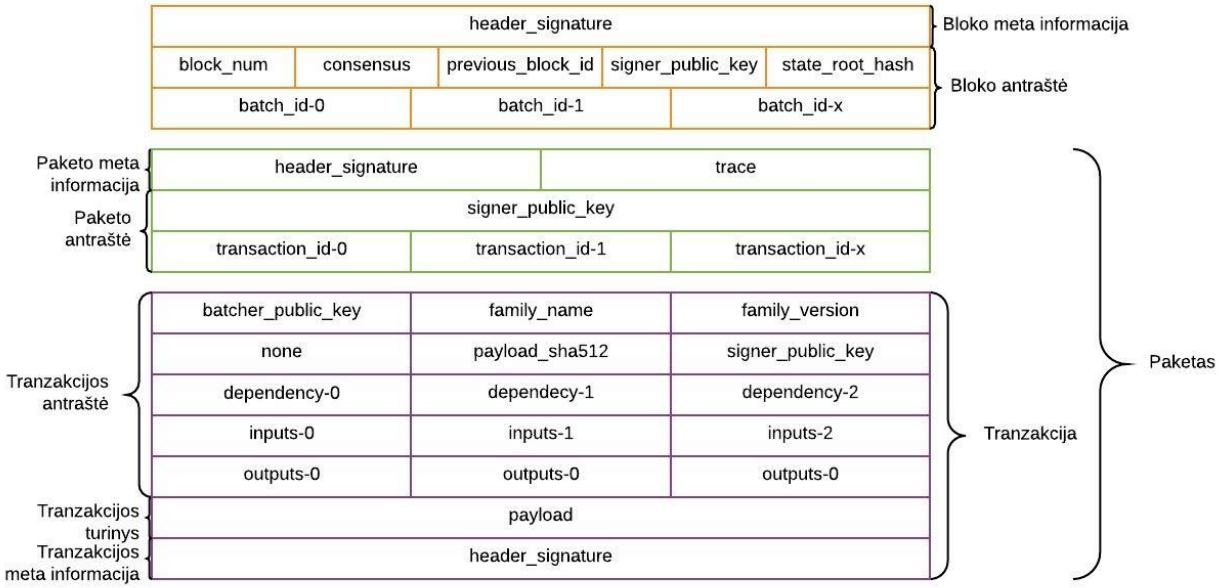
Atlikus pirminę autentifikaciją „RabbitMQ“ serveryje užklausa ištransliuojama klausytojams AMQPS protokolu, kurie įjungti blokų grandinės kliento dalyje. Blokų grandinės klientui gavus žinutę pradedamas antrasis autentifikacijos žingsnis, kuriame įrenginio identifikacinis numeris bus

tikrinamas blokų grandinės tinkle. Taigi pirmiausia būtina apibrėžti kaip turi vykti antrinė autentifikacija:

1. blokų grandinės „RabbitMQ“ klausytojo klientas gauna žinutę
2. prieš atliekant bet kokį kitą žingsnį – iš žinutės savybių paimamas siuntėjo identifikacinis numeris
3. paimta reikšmė perduodama blokų grandinės validatoriui su užklausa patikrinti reikšmę blokų grandinėje
4. Validatorius reikšmę patikrina globalioje būsenoje
5. jeigu reikšmė rasta – autentifikacija sėkminga

Analizės 1.3.3 skyriuje išsiaiškinta paketo bei tranzakcijos struktūra naudojant „Hyperledger Sawtooth“ blokų grandinių platformą bei kaip ji suformuojama validatoriuje. Šiame etape būtina detalizuoti kaip informacija bus perduodama ir išsaugojama įrenginio autentifikacijos ir registracijos metu.

Kadangi „Sawtooth“ tranzakcijos yra naudojamos modifikuoti globalią būseną kartu išsaugant duomenų vientisumą bei apsaugant nuo informacijos klastojimo, įrenginio registravimas reikš globalios būsenos pasikeitimą, o autentifikavimas – globalios būsenos patikrinimą, taigi tik registravimo metu bus sukuriamas paketas su tranzakcija ir perduodamas blokų grandinių tinklui. Norint įrodyti saugomų duomenų saugumą bei vientisumo užtikrinimą būtina detalizuoti pasirinktos blokų grandinių platformos validuoto bloko struktūrą, kuri pavaizduota paveikslėlyje 15. Pateiktoje struktūroje blokas turi meta informaciją, kurioje saugomas viso bloko antraštės parašas *header_signature*, pasirašytas bloko kūrėjo privačiuoju raktu naudojant *secp256k1* kreivę, o pati antraštė saugo informaciją apie bloko numerį *block_num* tinkle, serializuotą consensuso informaciją užkoduotą base64 formatu (*consensus*), prieš tai buvusio bloko identifikatorių *previous_block_id*, bloko pasirašytojo viešąjį raktą *signer_public_key*, globalios būsenos esamą adresą nuo pat šaknies (*state_root_hash*) bei sąrašą paketų identifikatorių (*batch_id-x*), priklausančių šiam blokui. Vėliau seka paketų sąrašas, kurių kiekvienas turi meta informaciją su 2 parametrais: paketo antraštės parašas pasirašytas su kūrėjo privačiu raktu bei sekimo (*trace*) ypatybė, nusakanti ar šis paketas turėtų būti sekamas per sistemą žurnalizavimo įrašais, taip sugeneruojant daugiau detalesnės informacijos, kuri reikalinga programėlės kūrimo etape. Paketo antraštė saugo paketo pasirašytojo viešąjį raktą bei sąrašą tranzakcijų identifikatorių *transaction_id-x* (antraščių parašų). Paketas gali turėti daugiau negu vieną tranzakciją su įvairiomis ypatybės, pavyzdžiui, tranzakcijos kūrėjo viešasis raktas *batcher_public_key*, tranzakcijos tvarkytojo šeimos pavadinimas (*family_name*), tranzakcijos tvarkytojo šeimos versija (*family_version*), atsitiktinės reikšmės skirtos užtikrinti parašo unikalumą (*nonce*), turinio maišos reikšmė SHA512 algoritmu (*payload_sha512*), tranzakcijos pasirašytojo viešasis raktas (*signer_public_key*), sąrašas privalomai pirmiau prieš šią tranzkaciją vykdomų tranzakcijų (*dependency-x*), sąrašas būsenos adresų kuriuos tranzakcija gali skaityti (*inputs-x*), sąrašas būsenos adresų į kuriuos tranzakcija gali rašyti (*outputs-x*). Toliau laikoma informacija apie tranzakcijos turinį *payload* užkoduota base64 formatu, o galiausiai tranzakcijos meta informacija laiko reikšmę apie tranzakcijos antraštės parašą, kuris pasirašytas su tranzakcijos kūrėjo privačiuoju raktu.



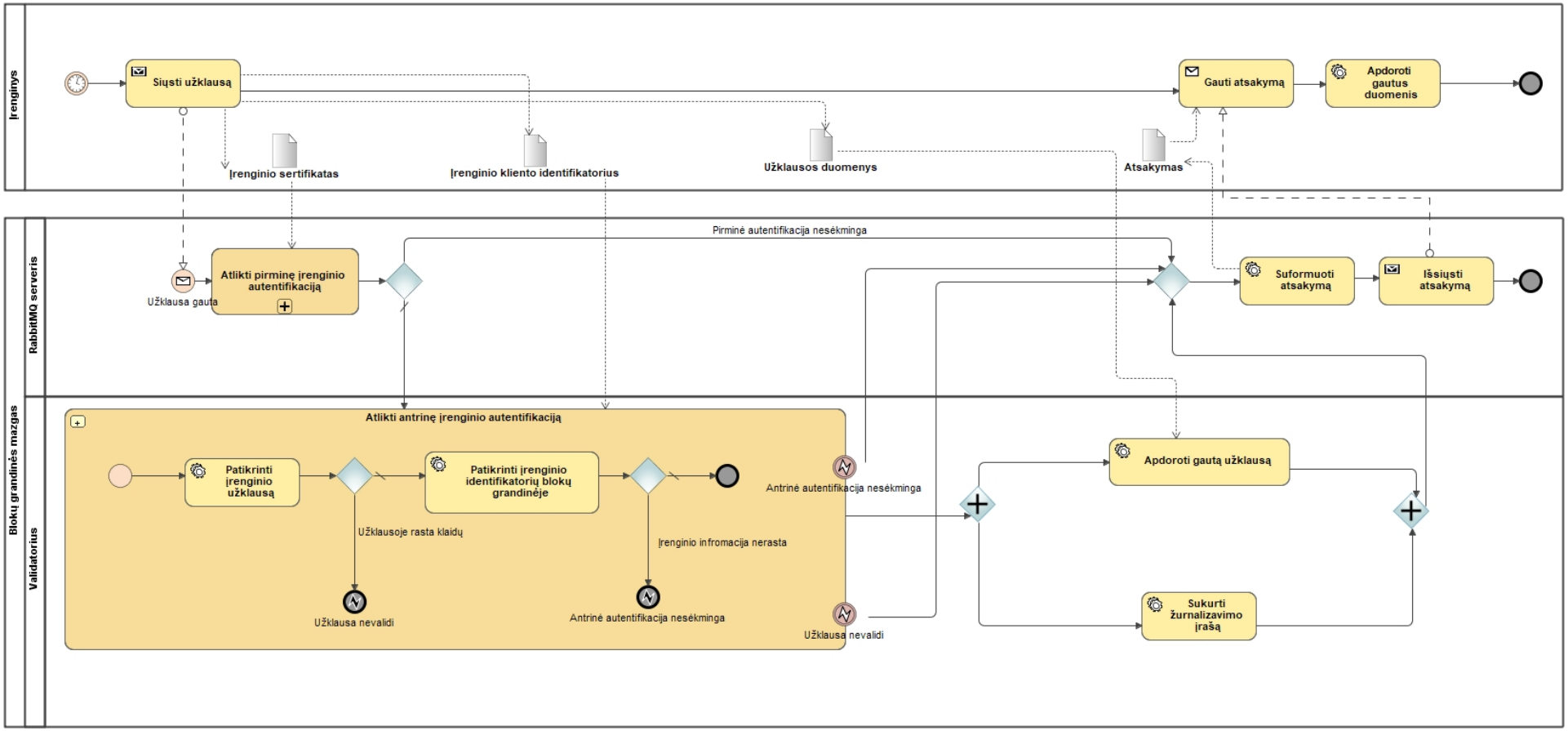
15 pav. „Hyperledger Sawtooth“ bloko struktūra

Taigi registruojant įrenginį blokų grandinėje bloko struktūroje jo informacija bus saugoma ne tik tranzakcijos turinyje, tačiau ir globalios būsenos užkoduotame adrese, o visas blokas apsaugotas nuo iškraipymų antraščių bei turinio parašais.

Autentifikuojant blokų grandinėse nebus reikalinga sukurti tranzakciją, kadangi autentifikuojant yra būtina tik patikrinti informaciją globalioje būsenoje, taigi validatorius duotą reikšmę patikrins su būsenoje esančiomis reikšmėmis.

2.1.4 Autentifikacijos veiklos diagrama

Išsiaiškinus abi autentifikacijos implementacijas buvo sukurta BPMN veiklos diagrama, kuri leistų detalizuoti planuojamą pilną autentifikavimo veiksmą. Įrenginio užklausos siuntimo ir autentifikacijos veiklos proceso modelio diagramoje (žr. pav. 16) procesas prasideda nuo įrenginio išsiųstos užklausos į blokų grandinės mazgo AMQP serverį. Gavus žinutę yra atliekama pirmoji įrenginio autentifikacija patikrinant turimus įrenginio programinės įrangos duomenis – klientui išduotus X.509 sertifikatus. Numatytuoju sėkmingu atveju pereinama į antrąją įrenginio autentifikacijos fazę, kur pirmiausia yra patikrinama įrenginio užklausa ir jos validumas, o vėliau – atliekamas įrenginio identifikacijos tikrinimas blokų grandinės tinkle. Jei įrenginys buvo užregistruotas blokų grandinėje ir pažymėtas kaip vis dar aktyvus – tuo atveju lygiagrečiai yra atliekami du uždaviniai: sukuriamas žurnalizavimo įrašas skirtas istorinei bei saugumo informacijai saugoti ir apdorojama gauta užklausa serverio programinės dalies. Tiek sėkmingu, tiek nesėkmingu autentifikacijos atveju – suformuojamas atsakymas ir išsiunčiamas įrenginiui, kuris galiausiai apdoroja gautus duomenis.



16 pav. Įrenginio užklausos siuntimo ir autentifikacijos veiklos proceso BPMN modelis

2.2 Daiktų interneto įrenginio registravimo/išregistravimo modelis

Dar viena labai svarbi modelio funkcija – įrenginių registravimo bei išregistravimo funkcionalumas, kurį galės atlikti tik administratorius, turintis blokų grandinėje blokų kūrimo teises. Registracijos modelis pateiktas paveikslėlyje 17. Mazgo administratoriui iniciavus naujo įrenginio sukūrimo užklausą, validatoriaus servisas pirmiausia inicijuos nurodytą tranzakcijų tvarkytoją, kuris pagal aprašytas taisykles bei logiką sukurs tranzakciją bei atliks lokalios būsenos adreso sukūrimą. Tada validatorius sudeda į paketą reikiamą informaciją (suformuotą tranzakciją, lokalios būsenos adresus) ir prideda pasirašytas reikšmes su savo privačiu raktu taip sukurdamas paketą, kurį pasirašo ir ištransliuoja į visą jo tinklą. „Sawtooth“ tinklo mazgai gavę paketą perduoda savo validatoriams, kurie validuoja ir pagal pakete nurodytas tranzakcijų šeimas, perduoda tranzakcijų tvarkytojams atlikti būsenos pakeitimus. Grąžinus tinklo atsakymą į paketo paskelbėjo validatorių – suformuojamas atsakymas administratoriaus mazgo klientui su sėkmingu arba klaidos statusu. Jeigu tranzakcija buvo priimta – ji yra įdedama į blokų grandinę ir nuo šiol įrenginys galės autentifikuotis pagal anksčiau aprašytą modelį (2.1). Analogiška seka vyksta ir norint išregistruoti įrenginį, tik tranzakcijų tvarkyklė gaus ne registravimo komandą, o išregistravimo, ko pasekoje globalioje būsenoje įrenginio identifikacinis numeris bei susijusi informacija bus ištrinta.

17 pav. Įrenginio registracijos veiklos UML diagrama

2.3 Daiktų interneto įrenginio autentifikavimo platformos projektavimo išvados

Apibendrinant realizacijos skyrių galima teigti, jog šiame skyriuje architektūriškai suprojektuotas sprendimas yra saugesnis nei analizės metu ištirti bei sumodeliuoti sistemos veikimo procesai, užtikrinantys saugų funkcionalumą, užtikrina saugų DI įrenginių autentifikavimą.

Realizacijos aukšto lygio diagramoje projektuojamas sprendimas išskirtas į 3 komponentus: išmanieji įrenginiai, blokų grandinės mazgas ir blokų grandinės tinklas. Norint įgyvendinti sprendimą išmanieji įrenginiai turi išpildyti šias sąlygas: mokėti susigeneruoti savo identifikacinį numerį, turėti išduotus sertifikatus, žinoti kaip suformuoti ir kur išsiųsti norimą informaciją. Tuo tarpu blokų grandinės mazgas yra kompleksiškiausias elementas, nes jis atlieka visą autentifikacijos logiką: priima žinutes iš įregistruotų įrenginių AMQPS protokolu per „RabbitMQ“ žinučių brokerį, atlieka įrenginio identifikatoriaus patikrinimą blokų grandinėje, suformuoja atsakymus, atlieka registraciją bei validaciją.

Būtina išskirti ir globalų įrenginio identifikacinį numerį, kuris sistemoje atliks šias funkcijas:

1. TLS sertifikatas bus išduodamas šiam identifikaciniam numeriui ir atitiks sertifikato vardą (CN)

2. „RabbitMQ“ serverio naudotojas bus registruojamas su šiuo identifikaciniu numeriu kaip prisijungimo vardas (angl. username)

3. „Hyperledger Sawtooth“ globalioje būsenoje įrenginys bus atpažinamas pagal šią reikšmę tiek registruojant įrengiinį, tiek autentifikuojant.

Pagrindinės suprojektuotos sistemos ypatybės yra 2 žingsnių autentifikacija, saugus komunikacijos kanalas tarp sistemos veikėjų bei sprendimo operavimas privačiame blokų grandinių tinkle. Implementuojant šį modelį pirminė autentifikacija atliekama susiejant įrenginį su blokų grandinės mazgo klientu apsaugotu AMQPS protokolu kartu su išduotais X.509 sertifikatais. Šis žingsnis yra labai svarbus, kadangi taip ne tik apsaugojamas komunikacijos kanalas tarp įrenginių ir blokų grandinės mazgų, tačiau kartu blokų grandinė yra apsaugoma nuo nepageidaujamos prieigos iš neregistruotų šaltinių, taip sumažinant riziką net ir patekus į tinklą – turėti galimybę bandyti įsilaužti į privatų blokų grandinės tinklą. Antrinis autentifikacijos žingsnis – įrenginio identifikatoriaus patikrinimas blokų grandinės globalioje būsenoje, kadangi blokų grandinė mums užtikrina duomenų vientisumą bei decentralizuotą informacijos saugojimą

3 Daiktų interneto įrenginių autentifikavimo sistemos prototipas

Norint įsitikinti suprojektuoto sprendimo realizavimo galimybėmis bei atlikti tyrimą būtina sukurti pirminį autentifikacijos metodo prototipą, panaudojant blokų grandinių technologiją. Šio prototipo realizacijai pasitelkta:

- „Docker“ [30] konteinerizacijos platforma – siekiant su kuo mažiau fizinių resursų įgyvendinti blokų grandinės tinklą. Kartu bus naudojamas „Docker Compose“ įrankis, palengvinantis konteinerių valdymą, pasirūpinantis tinklo apribojimais bei konteinerių nustatymais
- „Hyperledger Sawtooth“ blokų grandinių platforma – analizės dalyje išnagrinėta bei pasirinkta platforma
- „RabbitMQ“ žinučių brokeris – atsakingas už pirminę autentifikaciją bei žinučių surinkimą ir ištransliavimą klausytojams
- „Python“ programavimo kalba – „Hyperledger Sawtooth“ tranzakcijų tvarkytojo implementacijai bei siuntėjų/gavėjų klientų skriptų implementacijai
- „Raspberry Pi“ vienos plokštės kompiuteris – daiktų interneto įrenginys, kuris bus autentifikuojamas. Jis turės temperatūros bei drėgmės jutiklį „DHT11“, kuris imituos išmaniojo įrenginio veikimą bei informacijos siuntimą. Realizacijai naudotas „RaspberryPi 3B“ modelis su „Raspbian“ operacine sistema
- „tls-gen“[34] įrankis – OpenSSL paremtas įrankis, sugeneruojantis savarankiškai pasirašytus X.509 sertifikatus
- „SQLite“ – be serverio operuojanti duomenų bazė.

„Docker“ infrastruktūra ir „RabbitMQ“ serveris kuriami „Windows 10“ operacinės sistemos kompiuteryje.

3.1 Daiktų interneto įrenginio klientas

Pirmiausia būtina paruošti išmanųjį įrenginį autentifikacijai. Išeities kode 1 pavaizduotas „Python“ programavimo kalba parašytas daiktų interneto įrenginio kliento *dht_sensor.py* ištrauka, kurioje:

1. perskaitomas *machine-id* failas bei sugeneruojamas įrenginio identifikatorius panaudojant HMAC-SHA256 kriptografinę funkciją (1-3 eilutės)
2. paruošiama AMQPS jungtis su duotais sertifikatais, kurie sugeneruojami „tls-gen“ įrankiu bei kaip jungtis prie „RabbitMQ“ serverio: IP adresas, prievadas (5-8 eilutės)
3. nustatoma kliento žinučių tapatybė, prisijungiama prie serverio bei išsiunčiama žinutė (10-15 eilutės).

1 išeities kodas. RPi AMQP kliento ištrauka

```
with open("/etc/machine-id", "r") as f:
    machine_id = f.read().rstrip()
device_id = hmac.new("control", msg=machine_id, digestmod=hashlib.sha256).hexdigest()

context = ssl.create_default_context(cafile="ca_certificate.pem")
context.load_cert_chain("client_certificate.pem", "client_key.pem")
ssl_options = pika.SSLOptions(context, "authentication-server")
rmq_conn_params = pika.ConnectionParameters(host='192.168.1.226', port=5671, ssl_options=ssl_options,
    virtual_host='/', credentials=pika.credentials.ExternalCredentials())

properties = pika.BasicProperties(app_id=device_id)
connection = pika.BlockingConnection(rmq_conn_params)
channel = connection.channel()
channel.queue_declare(queue='master')
channel.basic_publish(exchange='', routing_key='master', body=message, properties=properties)
connection.close()
```

Taigi konkrečiu atveju *device_id* reikšmė išmanaus įrenginio yra *d6f3bc9c68718e2a90fbf13c1f1f5acd5092237e884a0bf1495107e0da5a026c*, prie kurios bus pririšama įrenginio tapatybė tiek pirmo, tiek antro autentifikacijos žingsnio metu.

3.2 Sertifikatų generacija

TLS sertifikatai reikalingi apsaugoti tinklą tarp blokų grandinės mazgo ir išmaniojo įrenginio bei pirminio atpažinimo moduliui. Įrenginio pavadinimas, kuriam generuojamas sertifikatas, priskiriamas kaip CN sertifikato reikšmė, kurią leidžia „tls-gen“ įrankis priskirti generavimo metu.

Paveikslėlyje 18 parodytas sugeneruotų sertifikatų sąrašas. Iš šio sąrašo juos galima suskirstyti į dvi grupes:

- „RabbitMQ“ serverio konfigūracijos – reikalingi *ca_certificate.pem*, *server_certificate.pem* bei *server_key.pem*
- daiktų interneto įrenginio kliento – reikalingi *ca_certificate.pem*, *client_certificate.pem* bei *client_key.pem*

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/30/2020   8:32 PM           1228 ca_certificate.pem
-a----         3/30/2020   8:32 PM           1736 ca_key.pem
-a----         3/30/2020   8:32 PM           1204 client_certificate.pem
-a----         3/30/2020   8:32 PM           2437 client_key.p12
-a----         3/30/2020   8:32 PM           1702 client_key.pem
-a----         3/30/2020   8:32 PM           1294 server_certificate.pem
-a----         3/30/2020   8:32 PM           2501 server_key.p12
-a----         3/30/2020   8:32 PM           1706 server_key.pem
```

18 pav. tls-gen įrankio sugeneruotų sertifikatų sąrašas

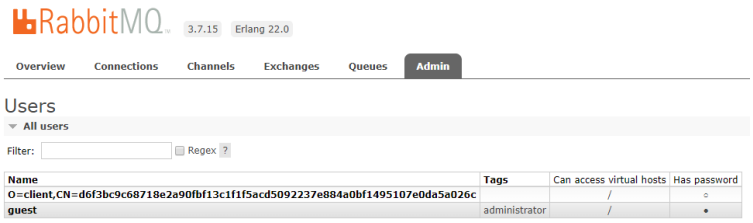
Pasirinktas įrankis be CN reikšmės leidžia konfigūruoti privataus rakto bitų skaičių, galiojimo laiką dienomis bei priskirti slaptažodį. Taigi įvykdžius komandą

make CN=d6f3bc9c68718e2a90fbf13c1f1f5acd5092237e884a0bf1495107e0da5a026c rezultatų kataloge yra sugeneruojami reikiami sertifikatai.

3.3 „RabbitMQ“ serverio įdiegimas

Tik įdiegus „RabbitMQ“ serverį pagal numatytuosius nustatymus – būtina pirmiausia užregistruoti naudotoją, skirtą išmaniojo įrenginio klientui, bei atlikti konfigūracinius pakeitimus, siekiant paruošti serverį TLS komunikacijai.

Paveikslėlyje 19 pavaizduotas naudotojų sąrašas „RabbitMQ“ administratoriaus puslapyje. *O=client,CN=d6f3bc9c68718e2a90fbf13c1f1f5acd5092237e884a0bf1495107e0da5a026c* tai naudotojo vardas, kuris priregistruotas sistemoje ir gali būti priskirtas išmaniojo įrenginio klientui, kadangi neturi prieigos prie kanalų, sukonfigūruotas tik rašymui vėduoklės principu bei neturi galimybės prisijungti su slaptažodžiu, nes nustatytas tik *EXTERNAL* galimas prisijungimo būdas (TLS sertifikatu), kuris taip pat užblokuoja galimybę prisijungti prie administravimo puslapio.



19 pav. RabbitMQ naudotojų sąrašas

Sukūrus naudotoją būtina taip pat užtikrinti, jog serveris leis prisijungti saugiu kanalu. Pagrindinė konfigūracija pavaizduota išeities kode 2. Kadangi numatytasis atidarytas prievadas AMQP protokolu „RabbitMQ“ serverio yra 5672 tai būtina pakeisti jį 5671 prievadu, kuris skirtas AMQPS komunikacijai. Tada nustatomas leistinas *EXTERNAL* autentifikacijos mechanizmas, taip įjungiant autentifikaciją pagal TLS sertifikatus. Paskutinis parametras *ssl_options* reikalauja nurodyti kelią iki serverio sertifikato failų vietos, kurie buvo sugeneruoti „tls-gen“ įrankiu. Papildomai yra nustatoma verifikuoti sertifikatą su *verify_peer* parametru bei neatlikti sėkmingos autentifikacijos (*fail_if_no_peer_cert*), jeigu klientas nepateikė sertifikatų.

2 išeities kodas. „RabbitMQ“ serverio konfigūracijos failas

```
1 [
2  {rabbit,
3   [
4     {ssl_listeners, [5671]},
5     {auth_mechanisms, ['EXTERNAL']}
6     {ssl_options, [
7           {cacertfile, "C:/Users/Justinas/AppData/Roaming/RabbitMQ/certs/ca_certificate.pem"},
8           {certfile, "C:/Users/Justinas/AppData/Roaming/RabbitMQ/certs/server_certificate.pem"},
9           {keyfile, "C:/Users/Justinas/AppData/Roaming/RabbitMQ/certs/server_key.pem"},
10          {verify, verify_peer},
11  {fail_if_no_peer_cert, true}]}]}
12 ].
```

Su šia „RabbitMQ“ serverio konfigūracija, sertifikatais bei išmaniojo įrenginio klientu platforma yra pasiruošusi atlikti pirminę autentifikaciją bei užtikrinti saugų ryšį AMQPS protokolu.

3.4 „Hyperledger Sawtooth“ blokų grandinių platformos įdiegimas

Kita prototipo įgyvendinimo dalis – antrojo autentifikavimo žingsnio implementacija su analizės dalyje pasirinkta „Hyperledger Sawtooth“ blokų grandinių platforma. Šią dalį galima išskirstyti į tris mažesnes dalis: platformos sukūrimas „Docker“ konteinerių aplinkoje, klientų bei tranzakcijos valdytojo sukūrimas ir prototipo paleidimas.

3.4.1 Platformos sukūrimas „Docker“ konteinerių aplinkoje

„Docker“ – tai OS lygio virtualizacija, kuri leidžia atskirti programėles ar programinę įrangą į pakuotes vadinamas konteineriais. Tai nėra atskiros virtualios mašinos, kadangi konteineriai operuoja ant pagrindinės mašinos operacinės sistemos kernelio, todėl resursų sunaudojimas bei jų reikalavimai yra ženkliai mažesni negu virtualių mašinų reikalaujami resursai, todėl norint sukurti prototipą, veikiantį ne viename blokų grandinės mazge, bet visame tinkle – „Docker“ virtualizacija buvo pasirinkta kaip vienas tinkamiausių sprendimų siekiant realizuoti prototipą.

Siekiant supaprastinti konfigūraciją bei kuriamos infrastruktūros valdymą buvo pasitelktas dar vienas įrankis – „Docker Compose“. Šis įrankis naudoja YAML tipo failus, kad nuskaičius failą sukonfigūruotų programos paslaugas ir jas startuotų. Prototipui skirto YAML failo ištrauka pavaizduota išeities kode 3 (pilnas konfigūracijos failas pateiktas priede H), kuriame aprašytas blokų grandinės tinklo validatoriaus konteineris:

- aprašomas konteinerio atvaizdas, pagal kurį bus startuojamas konteineris (2 eilutė). Šiuo atveju naudojamas „Hyperledger Sawtooth“ pateikiamas oficialus ir viešai platinamas atvaizdas
- nustatomas konteinerio pavadinimas (3 eilutė)
- aprašomi kokie prievadai bus atidaryti į tinklą (eilutės 4-7)
- nustatomas katalogas, kuris bus dalinamasis tarp konteinerio ir pagrindinės mašinos, taigi net ištrynus ar sukūrus konteinerį iš naujo – informacija šitame kataloge nebus prarasta (9 eilutė)
- aprašoma konteinerio paleidimo komanda, kuri reikš kol nesibaigs šita komanda – tol ir konteineris nesustos (eilutės 10-23). Ši ilga komanda pirmiausia sugeneruoja šio validatoriaus privatų ir viešą raktus, sukuria katalogą, kuriame jie bus laikomi, ir galiausiai startuoja „Sawtooth“ validatoriaus servisą nurodant prie kurių prievadų prijungti tinklo (8800), konsensuso (5050) bei komponento (4004) paslaugas. Taip pat nustatomi adresai kitų tinklo narių.

3 išeities kodas. „Docker Compose“ konfigūracijos failo ištrauka

```
1  validator−4:
2    image: hyperledger/sawtooth−validator:chime
3    container_name: sawtooth−validator−default−4
4    expose:
5      − 4004
6      − 5050
7      − 8800
```

```
 8    volumes:
 9      - poet-shared:/poet-shared
10    command: |
11      bash -c "
12        sawadm keygen --force && \
13        mkdir -p /poet-shared/validator-4 && \
14        cp -a /etc/sawtooth/keys /poet-shared/validator-4/ && \
15        sawtooth-validator -v \
16              --bind network:tcp://eth0:8800 \
17              --bind component:tcp://eth0:4004 \
18              --bind consensus:tcp://eth0:5050 \
19              --peering static \
20              --endpoint tcp://validator-4:8800 \
21              --peers tcp://validator-0:8800,tcp://validator-1:8800,tcp://validator-2:8800,tcp://validator
                    -3:8800 \
22              --scheduler parallel \
23              --network-auth trust
24      "
25    environment:
26      PYTHONPATH: "/project/sawtooth-core/consensus/poet/common:\
27        /project/sawtooth-core/consensus/poet/simulator:\
28        /project/sawtooth-core/consensus/poet/core"
29    stop_signal: SIGKILL
```

Tokiu formatu aprašomas kiekvienas platformos servisas, kuris reikalauja būti atskirtas. Vieną blokų grandinės mazgą sudaro šie konteineriai (*n* nurodo blokų grandinės mazgo numerį, kuriam ši paslauga priklauso ir yra susieta):

- *validator-n* – *n*-ojo blokų grandinės mazgo validatorius
- *rest-api-n* – REST sąsaja
- *control-tp-n* – daiktų interneto įrenginių autentifikavimo tranzakcijų valdytojas
- *settings-tp-n* – tvarko blokų grandinės konfigūravimo nustatymus, pavyzdžiui, konsensuso konfigūraciją, įjungtas tranzakcijų šeimas ir t.t.
- *poet-engine-n* – atsakingas už PoET logiką bei saugo informaciją apie patvirtintus blokus
- *poet-validator-registry-tp-n* – konfigūruoja PoET konsensuso įdiegimą ir validatorius.

Papildomai aprašomas dar vienas konteineris – *shell*, kuris atliks pirmojo blokų grandinės mazgo kliento vaidmenį ir tarpininkaus tarp „RabbitMQ“ serverio ir blokų grandinės. Prototipo realizacijai pasirinkta sukurti tinklą iš 5 mazgų, taip siekiant įsitikinti informacijos perdavimo bei tranzakcijų patvirtinimo greičiu. Taigi iš viso prototipą sudaro 5 blokų grandinės mazgų klientai bei iš viso 31 „Docker“ konteineris.

3.4.2 Kliento programėlės bei tranzakcijų tvarkytojo programėlės sukūrimas

Sukūrus bei aprašius blokų grandinės infrastruktūrą pradėta kurti blokų grandinės kliento programėlė bei tranzakcijos tvarkytojo programa. Šioms programėlėms sukurti pasirinkta „Python“

programavimo kalba, nes „Hyperledger Sawtooth“ turi šios kalbos paruoštas bibliotekas, pati programavimo kalba yra lengvai skaitoma, turi didelę bendruomenę bei nereikalingi papildomi kompiliatoriai, kadangi „Python“ programavimo kalba yra interpretuojama, o „Python“ interpretatorius yra integruotas į daugumą „Linux“ operacinės sistemos distribucijų.

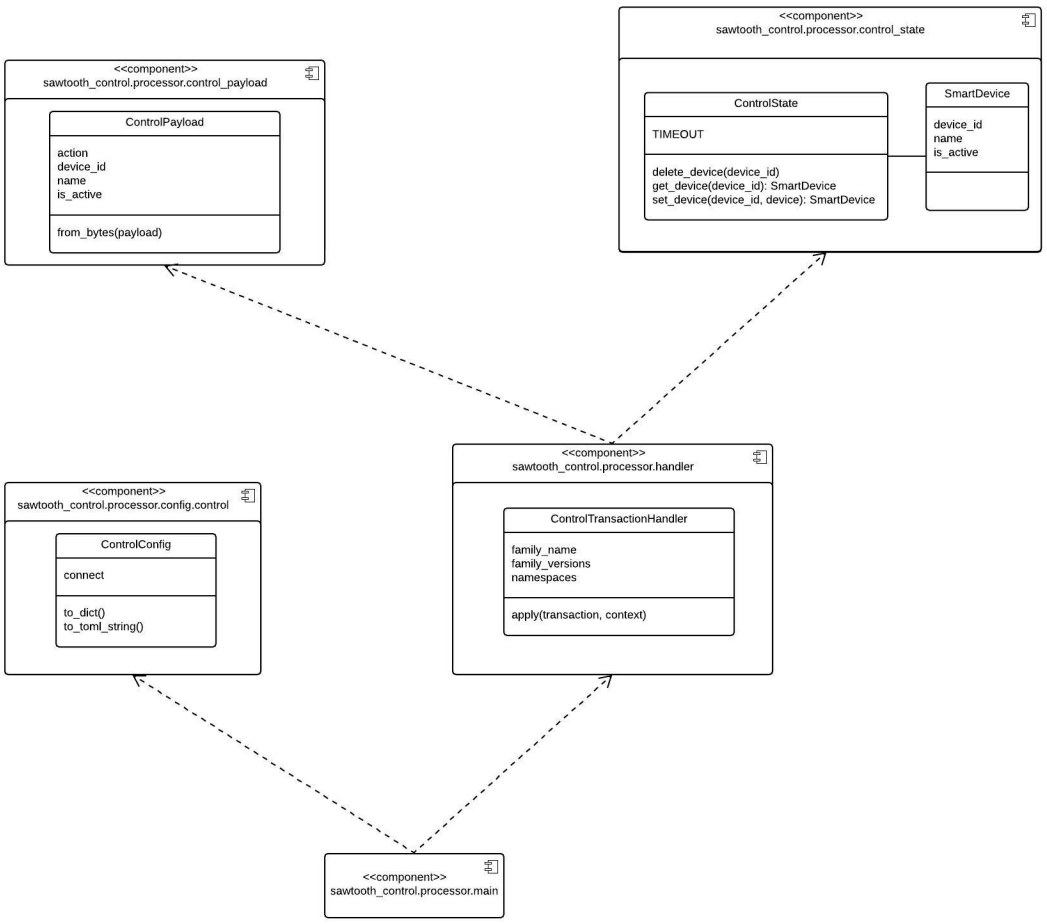
Pirmiausia sukurta tranzakcijos tvarkytojo programa *control-tp-python* (priedai D, E, F ir G), kurios pagrindinė funkcija – validuoti tranzakcijos duomenis bei valdyti nustatytos tranzakcijų šeimos (prototipo kūrimo ši šeima pavadinta „control“) globalią būseną. Paveikslėlyje 20 pavaizduota sukurto tranzakcijų tvarkytojo programėlės šaltinio kodo komponentų diagrama. Aukščiausio lygio komponentas yra *sawtooth_control.processor.main*, nuo kurio startuoja programa. Šis komponentas atsakingas už programėlės startavimo konfigūraciją bei nustato kokia eilės tvarka bus inicijuojama jungtis su pagrindiniu validatoriumi, kurio adresas reikalaujamas įvedimo argumentuose. Šis komponentas priklauso nuo *sawtooth_control.processor.config.control* komponento, turinčio *ControlConfig* klasę ir atsakingo už pradinės konfigūracijos išsaugojimą bei įkėlimą, ir *sawtooth_control.processor.handler* komponento. Pastarasis yra svarbus komponentas, kadangi turi *ControlTransactionHandler* klasę, kuri nusako tranzakcijų šeimos vardą, versiją bei programėlės logiką, kuri patikrinama prieš sukuriant tranzakciją. Diagramoje taip pat pavaizduota *sawtooth_control.processor.handler* komponento priklausomybė nuo *sawtooth_control.processor .control_payload* bei *sawtooth_control.processor.control_state* komponentų. *sawtooth_control .processor.control_payload* komponentas turi *ControlPayload* klasę, kuri yra tranzakcijoje užkoduoto objekto reprezentacija, o prototipo implementacijos metu nuspręsta, jog įrenginį apibrėš šios ypatybės:

- *action* – veiksmas, kuris atliekamas su įrenginiu. Siekiant parodyti funkcionalumą implementuotas tik sukūrimo (*create*) ir ištrynimo (*delete*) veiksmas
- *device_id* – įrenginio globalus identifikatorius
- *name* – įrenginio pavadinimas, siekiant palengvinti įrenginių atpažinimą platformos administratoriui
- *is_active* – ypatybė, žyminti ar įrenginys gali būti autentifikuojamas. Taip sukuriama galimybė laikinai atjungti įrenginį nuo sistemos.

Už globalios būsenos suformavimą bei valdymą atsakingas *sawtooth_control.processor.control_state* komponentas, turintis *ControlState* klasę, kuri atsakinga už veiksmų atlikimą būsenoje – sukurti įrašą, ištrinti įrašą, grąžinti būsenos įrašą. Pastaroji klasė susijusi su *SmartDevice* klase, kuri nusako kokia struktūra informacija bus grąžinama klientams. Būtina paminėti, jog implementacijai panaudota oficiali „Python“ kalbos „Hyperledger Sawtooth“ biblioteka [36].

Kliento programa išskaidyta į dvi programėles: blokų grandinės kliento bei tarpininko kliento programėles. Šio išskaidymo tikslas logiškai atskirti funkcijas į skirtingas programėles, siekiant lengvesnio palaikymo bei užtikrinti perpanaudojamumo galimybes:

1. blokų grandinės kliento programėlė (*control_cli.py* priedas B ir *control_client.py* priedas C) – skirta užtikrinti jungtį su validatoriumi per REST API servisą. Pagrindinė logika šioje programėlėje – tai įvesties tikrinimas ir tranzakcijų ir paketų sukūrimas bei nusiuntimas į priskirtą REST API servisą validatoriui. Tranzakcijos bei paketo formavimas pavaizduotas



20 pav. „control“ šeimos tranzakcijų tvarkytojo programėlės komponentų diagrama

išeities kode 4. *ControlClient* klasės privati funkcija *_send_device_txn* pirmiausia aprašo tranzakcijos turinį 2-oje eilutėje – tai simbolių eilutė, kurioje kableliu atskirtos įrenginio vardas, veiksmas, globalus įrenginio identifikatorius bei *is_active* žymė. Šios reikšmės yra tos pačios, kurios sudaro struktūrą tranzakcijos tvarkytojo struktūroje. Eilutės 6-16 sukuria tranzakcijos antraštę su aprašytais parametrais, 18-oje eilutėje atliekamas pasirašymas, o eilutėse 20-23 atliekamas tranzakcijos iniciavimas. Galiausiai 25 eilutė sukuria paketą bei jo identifikatorių eilutėje 26. Verta paminėti, jog ši struktūra yra įgyvendinama pagal analizėje ištirtą struktūrą 1.3.3.

4 išeities kodas. Blokų grandinės kliento programėlės tranzakcijos bei paketo formavimas

```
1 def _send_device_txn(self, name, action, device_id="", is_active="", wait=None):
2     payload = ",".join([name, action, device_id, is_active]).encode()
3
4     address = self._get_address(device_id)
5
6     header = TransactionHeader(
7         signer_public_key=self._signer.get_public_key().as_hex(),
8         family_name="control",
9         family_version="1.0",
```

```
10              inputs=[address],
11              outputs=[address],
12              dependencies=[],
13              payload_sha512=_sha512(payload),
14              batcher_public_key=self._signer.get_public_key().as_hex(),
15              nonce=hex(random.randint(0, 2**64))
16      ).SerializeToString()
17
18      signature = self._signer.sign(header)
19
20      transaction = Transaction(
21              header=header,
22              payload=payload,
23              header_signature=signature)
24
25      batch_list = self._create_batch_list([transaction])
26      batch_id = batch_list.batches[0].header_signature
```

2. tarpininko kliento programėlė (*collector.py* priedas A) – atsakinga už sąsają tarp „RabbitMQ“ serverio bei blokų grandinės mazgo. Išeities kode 5 pavaizduota tarpininko programėlės pagrindinės logikos ištrauka. Šioje ištraukoje pirmoje eilutėje konfigūruojama jungtis su „RabbitMQ“ serveriu, 17 eilutė nusako kokios eilės bus klausomasi bei nusako, jog gavus žinutę ji bus perduodama *callback* funkcijai, o 18 eilutė prisijungia prie serverio ir pradeda žinučių prenumeraciją. Gavus žinutę ir perdavus ją į nurodytą funkciją įvyksta antrasis autentifikavimo žingsnis ir iškviečiamas blokų grandinės kliento programėlė su *show* argumentu, kuris atlieka patikrinimą blokų grandinės globalioje būsenoje pagal globalų įrenginio identifikatorių, kuris buvo perduotas per žinutės ypatybes *properties*. Jeigu blokų grandinės kliento programėlė grąžino teigiamą atsakymą – reiškia autentifikacija blokų grandinėje buvo sėkminga ir įrenginys yra autentifikuotas. Taigi įrenginio siųsti temperatūros duomenys yra išsaugomi į „SQLite“ duomenų bazę (15 eilutė)

5 išeities kodas. Tarpininko kliento programėlės ištrauka

```
1 conn_params = pika.ConnectionParameters(host="host.docker.internal", port=5671, virtual_host="/",
        ssl_options=ssl_options, credentials=pika.credentials.ExternalCredentials())
2
3 connection = pika.BlockingConnection(conn_params)
4 channel = connection.channel()
5 channel.queue_declare("master")
6
7 def callback(ch, method, properties, body):
8       process = subprocess.run(['control', 'show', '--url', 'http://sawtooth-rest-api-default-0:8008',
              properties.app_id], stdout=subprocess.PIPE, universal_newlines=True)
9       try:
10              data = json.loads(process.stdout)
11      except json.JSONDecodeError as e:
12              data = None
13
14      if data and data['is_active'] == "true":
```

```
15            insert_data(db_conn, (data['device_id'], body.decode('utf−8'), data['name']))
16
17 channel.basic_consume(queue="master", on_message_callback=callback, auto_ack=True)
18 channel.start_consuming()
```

Kliento dalys išskaidytos siekiant atskirti su blokų grandinės autentifikavimo logika susijusį kodą nuo verslo logikos, susijusios su konkrečia išmaniųjų įrenginių grupe. Pavyzdžiui, turint temperatūros daviklių išmaniųjų įrenginių bei judesio jutiklių – blokų grandinės kliento programėlė, atliekanti autentifikaciją, bus ta pati, o tarpininko kliento programėlės bus skirtingos kiekvienai grupei, kadangi jos aprašys logiką kaip duomenys po autentifikacijos bus tvarkomi.

Sukūrus blokų grandinės kliento, tarpininko kliento bei tranzakcijos tvarkytojo programėles – prototipas yra paruoštas paleidimui bei išmaniojo daiktų įrenginio autentifikavimui.

3.4.3 Prototipo paleidimas ir informacijos srautas

Prototipo aplinkos paruošimas įvykdomas atlikus šias komandas:

1. Blokų grandinės platformos paleidimas – komanda *docker-compose -f sawtooth-default-poet.yaml up*

2. „RabbitMQ“ serverio paleidimas bei 2 klientų užregistravimas: *d6f3bc9c68718e2a90fbf13c1f1f5acd5092237e884a0bf1495107e0da5a026c* – išmaniojo įrenginio kliento, *DESKTOP-O3O8HC4* – blokų grandinės tarpininko kliento

3. „RaspberryPI“ vienos plokštės kompiuterio su temperatūros davikliu įjungimas bei įkėlimas paruoštos programos su sertifikatais

Paskutinis reikalingas žingsnis – įrenginio užregistravimas su komanda *control create RPI-device d6f3bc9c68718e2a90fbf13c1f1f5acd5092237e884a0bf1495107e0da5a026c true –-url http://sawtooth-rest-api-default-0:8008*, kuri perduos blokų grandinės validatoriui sukurti įrenginį blokų grandinėje su vardu „RPI-device“, kurio globalus identifikatorius yra jau anksčiau išsiaiškintas RPi įrenginio identifikatorius, be to nustatoma, jog šis įrenginys yra aktyvus ir paruoštas autentifikacijai (žymė *true*), o su *–-url* žyme nurodome į kurį blokų grandinės mazgą kreipiamės. Sėkmingai įvykdžius šią komandą yra sukuriama tranzakcija ir paskelbiama tinklui apie atnaujintą globalią būseną. Įsitikinti sėkminga registracija galima patikrinus blokų grandinėje esančias tranzakcijas – *sawtooth block list –-url http://sawtooth-rest-api-default-0:8008* grąžins blokų sąrašą, o patikrinti paskutinio bloko informaciją galima komanda *sawtooth block show –-url http://sawtooth-rest-api-default-0:8008 8d1dd958c9cd2d1fe2f<...>439906612bff7cdffc972*, kurios rezultatas pavaizduotas paveikslėlyje 21.

Šiame paveikslėlyje pavaizduota bloko struktūra atitinka anksčiau 2.1.3 išsiaiškintą bloko struktūrą, paveikslėlyje esančio parametro *payload* reikšmę *UlBJLWRldmljZSxjcmVhdGUsZDZmM2JjOWM2ODcxOGUyYTkwZmJmMTNjMWYxZjVhY2Q1MDkyMjM3ZTg4NGEwYmYxNDk1MTA3ZTBkYTVhMDI2Yyx0cnVl* dekodavus iš base64 formato – gaunamas *RPI-device,create,d6f3bc9c68718e2a90fbf13c1f1f5acd5092237e884a0bf1495107e0da5a026c,true* rezultatas. Ši simbolių eilutė patvirtina, jog tranzakcija sėkmingai užregistruota blokų grandinėje ir globalioje būsenoje.

21 pav. Įrenginio registracijos metu sukurto bloko informacija

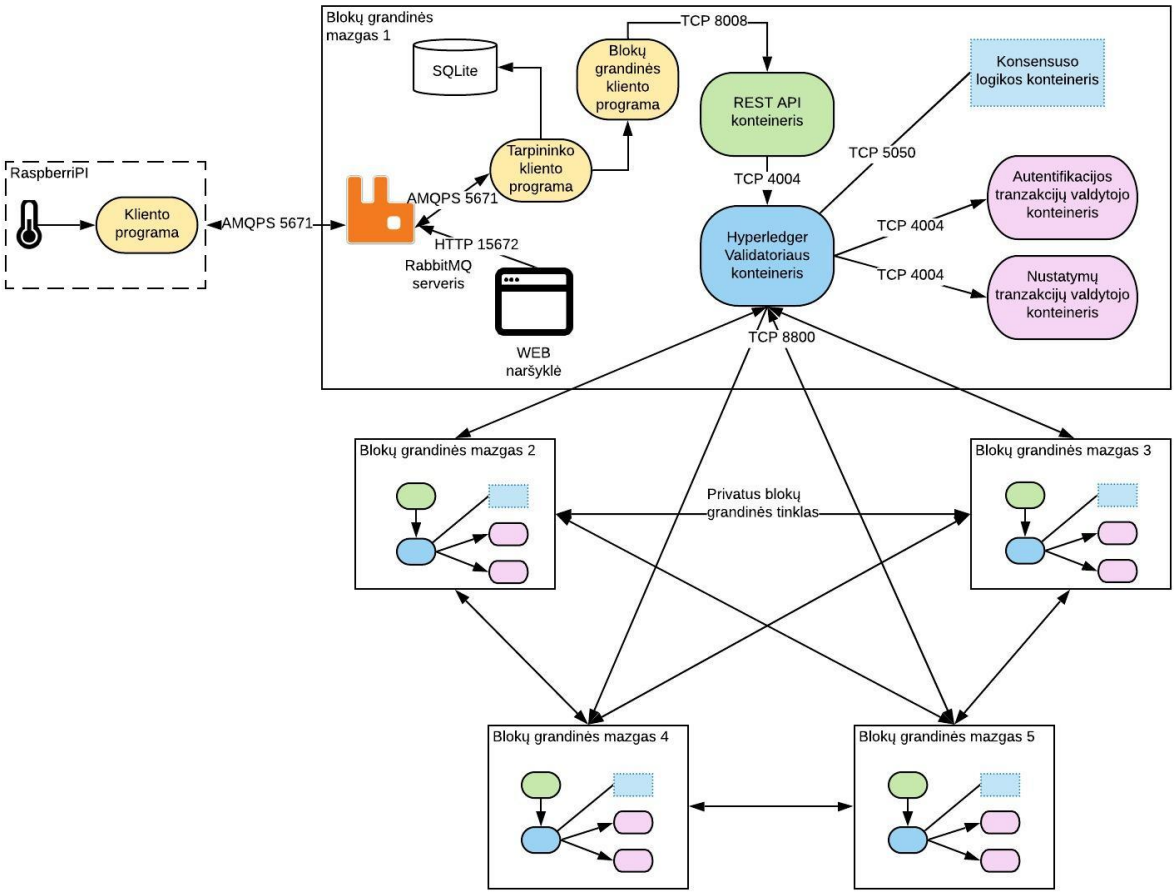
Atlikuos šiuos žingsnius aplinka yra paruošta atlikti suprojektuotą autentifikaciją. Pirmiausia startuoja *collector.py* scenarijus ir prisijungia prie „RabbitMQ“ serverio AMQPS protokolu ir pradeda klausyti žinučių. Tada iš vienos plokštės kompiuterio paleidžiamas *dht_sensor.py* scenarijus, kuris gauna informaciją iš temperatūros jutiklio ir informaciją išsiunčia į „RabbitMQ“ serverį. Tik išsiuntus informacija gauta tarpininko kliente, tai reiškia pirminė autentifikacija atlikta sėkmingai, tada atliekama antrojo lygio autentifikacija blokų grandinėje *control show –-url http://sawtooth-rest-api-default-0:8008 d6f3bc9c68718e2a90fbf13c1f1f5acd5092237e884a0bf1495107e0da5a026c* ir gavus teigiamą atsakymą iš blokų grandinės kliento – temperatūros informacija įterpiama į „SQLite“ duomenų bazę.

3.5 Prototipo realizacijos išvados

Apibendrinant galima teigti, jog remiantis analizės metu surinkta informacija bei projektavimo metu sukurta struktūra buvo sėkmingai sukurtas veikiantis prototipas, įrodantis suprojektuoto autentifikavimo modulio galimybes bei saugumo charakteristikas. Paveikslėlyje 22 pavaizduota detali suprojektuotos platformos schema, kurioje pavaizduoti visi šiame skyriuje sukurti bei aprašyti platformos komponentai:

- „RaspberryPi“ vienos plokštės kompiuteris, kuris platformoje atliko išmanaus įrenginio vaidmenį, su kliento programa
- Blokų grandinės mazgas, sudarytas iš:
  - „RabbitMQ“ serverio
  - tarpininko kliento programos
  - blokų grandinės kliento programos
  - REST API serviso
  - „Hyperledger Sawtooth“ validatoriaus
  - konsensuso logikos konteinerio
  - autentifikacijos tranzakcijų valdytojo
  - tinklo nustatymų tranzakcijų valdytojo

- Privatus blokų grandinės tinklas



22 pav. Prototipo realizacijos schema

Sukurta platforma išpildo užsibrėžtus reikalavimus projektavimo žingsnyje 2.3 iškeltus reikalavimus: išskirtas globalus įrenginio identifikacinis numeris, įrenginys su žinučių brokeriu bendrauja TLS protokolu apsaugotu kanalu per AMQPS protokolą, autentifikavimas atliekamas 2 žingsniais, blokų grandinė veikia privačiame tinkle.

Būtina paminėti, jog sukurta platformos architektūroje autentifikacija yra visiškai atsieta nuo daiktų interneto įrenginio verslo logikos, taigi infrastruktūra gali būti plečiama pridedant įvairių kitų išmaniųjų įrenginių, tokiu atveju užtenka aprašyti tarpininko kliento programas su reikiama verslo logika.

4 Daiktų interneto įrenginių autentifikavimo sistemos prototipo eksperimentinis tyrimas

Norint įsitikinti sumodeliuoto metodo bei sukurto prototipo charakteristikomis – būtina atlikti kokybinį bei kiekybinį sprendimo eksperimentinį tyrimą. Kiekybinis tyrimas atliktas panaudojant prototipo techninę bazę:

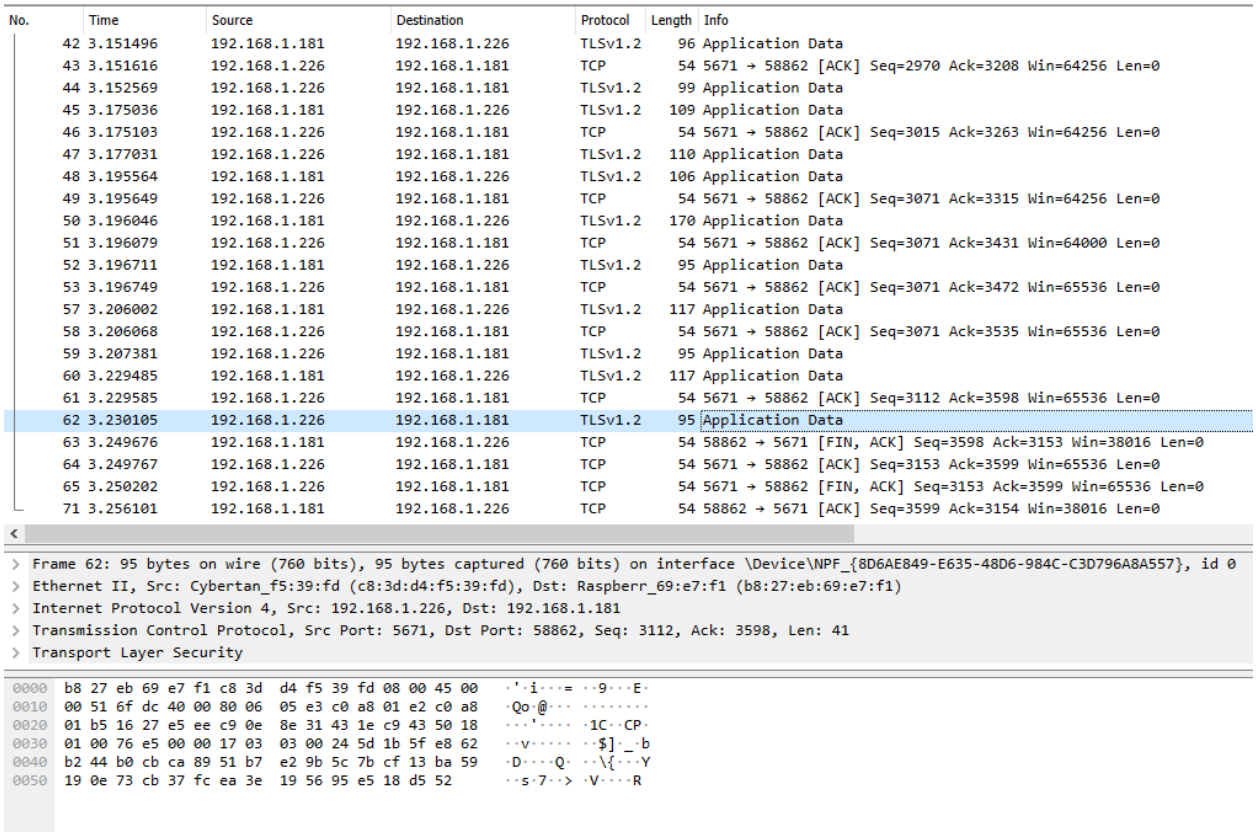
- Lenovo nešiojamas kompiuteris su Intel Core i7-7700HQ tipo 2.8GHz procesoriumi, 8GB operatyviosios atminties bei „Windows 10 Pro“ operacine atmintimi. Šiame kompiuteryje veikia blokų grandinės tinklas bei žinučių brokerio serveris.
- „RaspberryPi 3B“ vienos plokštės kompiuteris su Broadcom 1.2GHz procesoriumi, 1GB operatyviaja atmintimi ir „Raspbian“ 2020-02 operacine sistema
- „DHT11“ temperatūros ir drėgmės jutiklis
- „Docker Desktop“ bendruomenės versija 2.2.0.4 su „Docker“ 19.03.8 bei „Docker Compose“ 1.25.4 versija
- „RabbitMQ“ žinučių brokerio serveris 3.7.15 versija
- „Hyperledger Sawtooth“ blokų grandinės platforma 1.2.4 versija
- „Python“ 3.8 programavimo kalba

4.1 Kokybinis sprendimo tyrimas

Prieš atliekant kiekybinį sprendimo tyrimą būtina detalizuoti kokybines sprendimo savybes, kurios išsprendžia analizės skyriuje 1.1 išskirtas pagrindines daiktų interneto įrenginių problemas.

Vientisumas – viena pagrindinių charakteristikų, kurią sprendimui suteikia blokų grandinių platformos panaudojimas. Analizuojant blokų grandines skyriuje 1.3.1 ši savybė apibūdinta kaip užtikrinanti nekintančius įrašus dėl kiekvieno bloko žinojimo apie prieš tai buvusio bloko šifro. Pasiūlyto metodo bei įgyvendinto prototipo kontekste (kaip išsiaiškinta skyriuje 3.4.3) vientisumas užtikrinamas ne tik prieš tai esančio bloko šifru, bet ir pateikiant paketo, tranzakcijos bei tranzakcijos turinio parašus, pasirašytus su siuntėjo privačiu raktu, taigi kiekvienas tinklo validatorius užtikrina, jog modifikuota informacija nesutampanti su atsiųstais parašais nepatektų į blokų grandinę. Tinklo validavimas taip pat užtikrina aukščiausio lygio duomenų validavimą, kadangi viso tinklo narių validatoriams patikrinus ir patvirtinus tranzakciją – įrenginys yra užregistruojamas blokų grandinėje.

Patobulinta ir saugesnė autentifikacija – ši charakteristika sprendžia saugumo bei identifikacijos problemą, daugybės autorių išskirtą vieną pagrindinių daiktų interneto įrenginių problemų. Pasiūlytas sprendimas tai užtikrina atlikdamas dviejų faktorių autentifikaciją, kai pirmasis autentifikacijos žingsnis yra atliekamas žinučių brokerio, o antrasis – blokų grandinės platformos. Būtina pabrėžti, jog pirmasis autentifikacijos žingsnis yra atliekamas patikrinant ne daiktui priskirtą prisijungimo vardą ir slaptažodį, tačiau pagal priskirtą TLS sertifikato identitetą, kurį suklastoti yra itin sudėtinga.



23 pav. Užšifruotas žinučių turinys tinkle

Privatumas – daugumoje nagrinėtų metodų ši savybė nebuvo apgalvota ar išskirta kaip viena svarbesnių. Sukurtas metodas siunčiamų duomenų privatumą užtikrina privačiu blokų grandinės sprendimu bei saugiu AMQPS protokolu, kuris užšifruoja duomenis TLS protokolu, taip užkertant kelią duomenų nutekėjimui, žinutėms keliaujant tarp daiktų interneto įrenginio ir žinučių brokerio, bei duomenų modifikavimui, smarkiai sumažinant MITM (angl. Man-in-the-middle) atakos riziką. Paveikslėlyje 23 matomos užšifruotos daiktų įrenginio siunčiamos AMQPS žinutės, keliaujančios saugiu TLS protokolu, kurių turinio negalima perskaityta net perėmus žinutę. Ši charakteristika išsprendžia S. Huh'o, S. Cho ir S. Kim'o darbe [16] išskirtas viešos blokų grandinės keliamas problemas: greitį bei privatumo trūkumą.

Decentralizuota sistema – visa informacija saugoma kiekviename blokų grandinės įrenginio kliente bei globalioje būsenoje, o tai užtikrina sistemos veikimą net praradus vieną ar daugiau narių, taigi platformoje nelieka pavienio pažaidos taško. Ši savybė sprendimui užtikriną beveik visišką apsaugą nuo DoS (angl. Denial of Service) bei DDoS (angl. Distributed Denial of Service) atakų, kadangi sutrikdžius tik vieno blokų grandinės tinklo kliento narį sistema gali visiškai operuoti toliau nepakitusi. Decentralizuota sistema išsprendžia X. Li, J. Niu ir kitų autorių darbe [13] išskirtą problemą – pavienį pažaidos tašką.

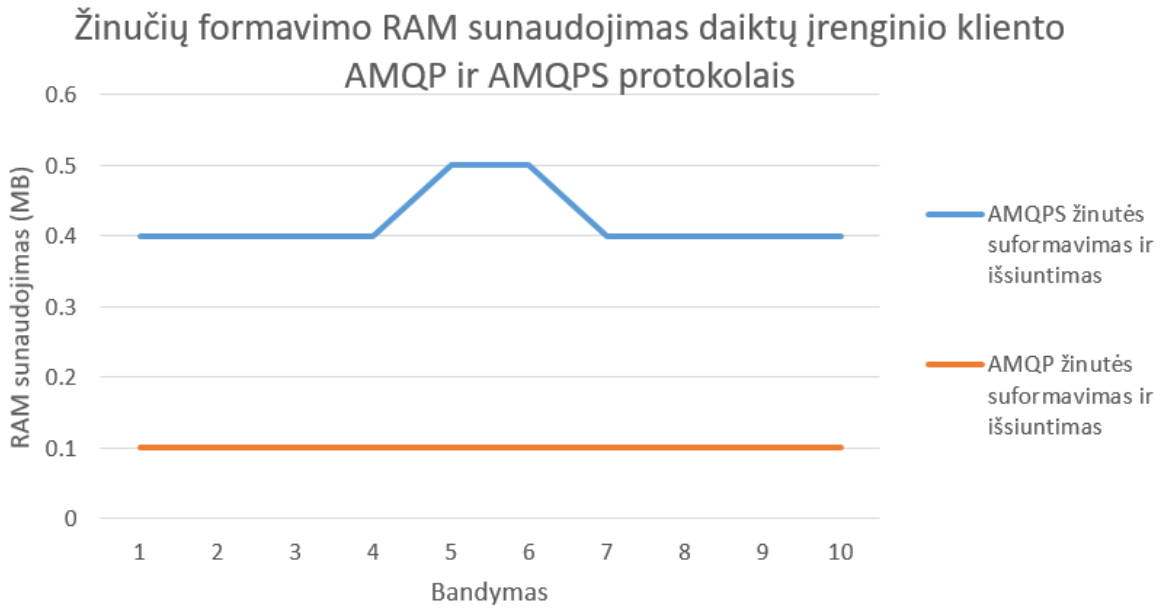
4.2 Kiekybinis sprendimo tyrimas

Kiekybinio tyrimo metu prototipo aplinka buvo modifikuojama bei didinama siekiant išsiaiškinti resursų sunaudojimą ir reikiamus laiko tarpus įvairiems veiksmams atlikti bei jų didėjimo ar mažėjimo tendencijas. Išskirti pagrindiniai sistemos komponentai, kurių ištyrimas yra būtinas šio

darbo perspektyvoje:

- daiktų įrenginio resursų sunaudojimą bei reikiamą laiką formuojant žinutes
- tarpininko kliento atsiradęs vėlinimas dėl antrojo autentifikacijos žingsnio

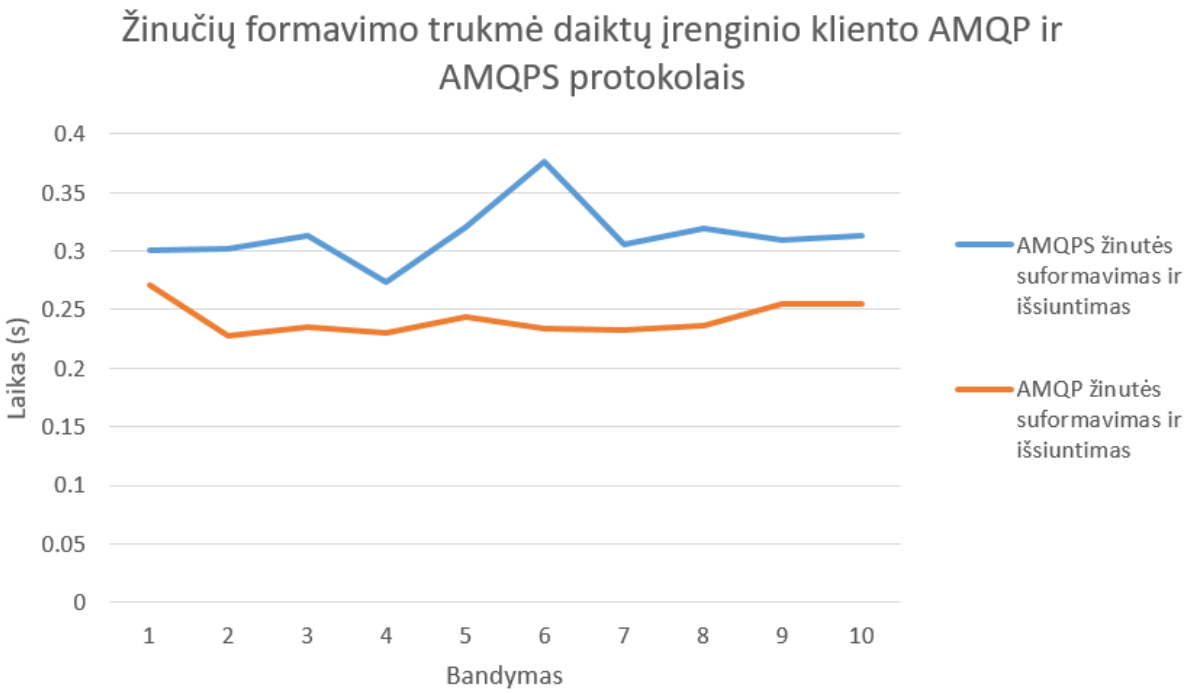
Sprendimo bei platformos tinklo pralaidumas bei vėlinimas nebus tiriamas, kadangi tai priklauso nuo tinklo jungčių, o prototipas buvo realizuotas viename kompiuteryje, taigi rezultatai būtų tik sąlyginiai, todėl nuspręsta orientuotis į metodo suteikiamų saugumo charakteristikų kainą laiko bei resursų atžvilgiu. Taip pat būtina paminėti, jog tyrimui buvo naudojamos *memory-profiler*[37] bei *timeit*[38] „Python“ kalbos bibliotekos.



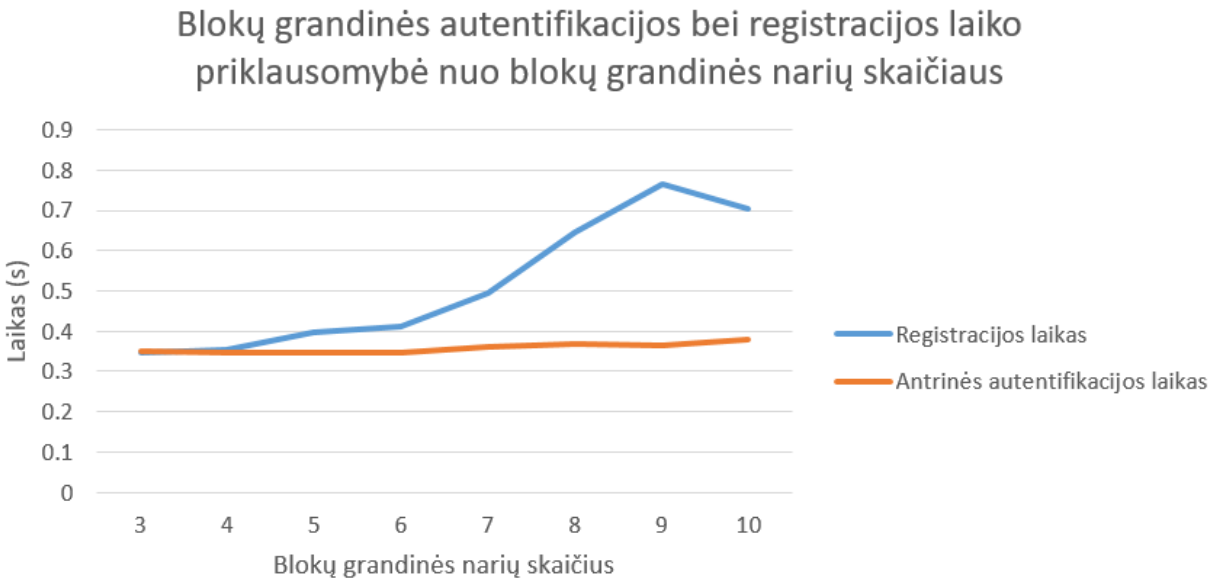
24 pav. Žinučių formavimo RAM sąnaudų palyginimas daiktų interneto įrenginyje

Pirmiausia pasirinkta ištirti įrenginio kliento operatyviosios atminties sunaudojimą formuojant AMQPS žinutę kartu suformuojant ir globalų identifikatorių bei palyginti šiuos rezultatus su AMQP žinutės suformavimu bei atliekant *PLAIN* autentifikaciją prisijungimo vardu bei slaptažodžiu taip nesuformuojant globalaus identifikatoriaus. Atlikti 10 bandymų, kurių rezultatai pavaizduoti diagramoje paveikslėlyje 24. Iš šios diagramos galima teigti, jog RAM sunaudojimas skiriasi ženkliai, kadangi formuojant AMQP žinutes ir atliekant bazinį autentifikavimo žingsnį su žinomais prisijungimo vardu bei slaptažodžiu rezultatai buvo stabilūs ir operatyviosios atminties sunaudojimas neviršijo 0.1 MB, o AMQPS žinutės suformavimas bei identifikatoriaus sukūrimas svyravo tarp 0.4 bei 0.5 MB. Bet kokiu atveju pastarieji skaičiai yra sąlyginai nedideli net daiktų interneto įrenginių kontekste.

Atlikus operatyviosios atminties analizę – ištirti laiko trukmės skirtumai taipogi tarp AMQPS ir AMQP žinutės suformavimo ir išsiuntimo daiktų interneto įrenginio kliente. Šio tyrimo metu irgi atlikti 10 bandymų, kurių rezultatai pateikti paveikslėlyje 25. Šioje diagramoje AMQPS žinutę su unikaliu mašinos identifikatoriumi suformuoti vidutiniškai prireikė 0.31 sekundės, o AMQP su numatytuoju autentifikacijos mechanizmu vidutiniškai 0.24 sekundės, taigi pasiūlymas sprendimas su globaliu įrenginio identifikatoriumi bei perduodant informaciją saugiu ryšiu vidutiniškai pareikalauja apie 30% daugiau laiko.



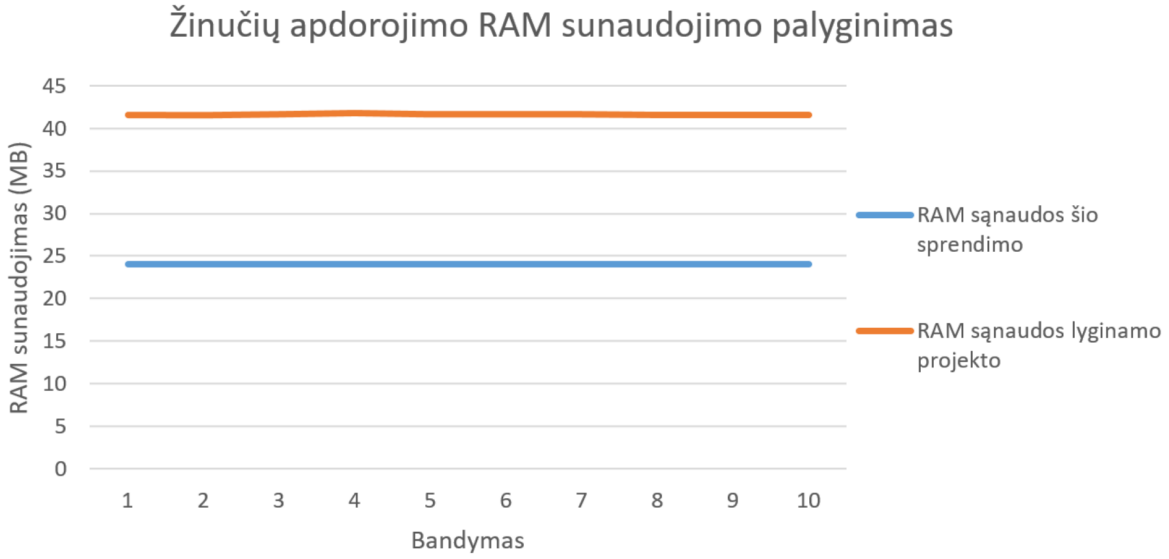
25 pav. Žinučių formavimo trukmės palyginimas daiktų interneto įrenginyje



26 pav. Autentifikacijos bei registracijos laiko priklausomybė nuo tinklo narių skaičiaus

Dar vienas svarbus tyrimas – blokų grandinės antrojo autentifikacijos žingsnio bei įrenginio registracijos laiko priklausomybė nuo blokų grandinės narių skaičiaus. Šio tyrimo metu platforma buvo perkuriama su vis didesniu narių skaičiu ir tiriamas įrenginio registracijos laikas bei vidutinis 10 bandymų antrinės autentifikacijos laikas. Matuojant laiką nebuvo įskaičiuojamas jungties sudarymo laikas bei prototipo programėlės logika (temperatūros duomenų patalpinimas į „SQLite“ duomenų bazę), kadangi tai nėra tyrimo subjektas ar dalis pasiūlyto metodo. Paveikslėlyje 26 pavaizduota diagrama su šio tyrimo rezultatais. Diagramoje vidutinis antrinės autentifikacijos laikas nesikeitė net ir didėjant blokų grandinės narių skaičiui, o tai patvirtina globalios būsenos komponentą „Hyperledger Sawtooth“ platformoje, taigi net ir didinant blokų grandinės narių skaičių

esamų klientų autentifikacijai tai nedarys įtakos. Laikas, reikalingas autentifikuoti įrenginį, vidutiniškai siekė 0.35 sekundes, tuo tarpu registracijos laikas proporcingai kilo augant blokų grandinės narių skaičiui nuo 0.35 sekundžių iki 0.7 sekundės, taigi didinant blokų narių skaičių – naujų įrenginių registracija linkusi taip pat ilgėti.



27 pav. Žinučių apdorojimo RAM sunaudojimo palyginimas

Paskutinis tyrimas atliktas siekiant palyginti sukurto sprendimo parodytus rezultatus su lyginamų eksperimentinių metodų atliktomis analizėmis. Palyginimui išrinktas eksperimentinis metodas, atlikęs tiksliausią analizę lyginant su kitais analizuotais darbais – Ž. Radzevičiaus tyrimas atliktas jo darbe [17] apie įrenginių autentifikavimą blokų grandinėse. Šiame tyrime autorius ištyrė kiek sukurtoje aplinkoje pareikalauja operatyviosios atminties žinučių apdorojimas gavus iš žinučių brokerio ir patikrinus įrašus blokų grandinėje. Atitinkamai pabandyta atkartoti tokį patį eksperimentą sukurto prototipo aplinkoje ir gavus rezultatus juos palyginti. Atlikus tyrimus gauti rezultatai pateikti paveikslėlyje 27 esančioje diagramoje, iš kurios galima matyti, jog šio darbo metu sukurtas sprendimas žinučių apdorojimui ir patikrinimui su blokų grandinėje esančiais duomenimis apytiksliai pareikalauja 17 MB mažiau operatyviosios atminties arba apie 40% mažiau RAM sąnaudų.

Būtina pabrėžti, jog šis kiekybinis tyrimas atliktas paleidžiant visus blokų grandinės narius viename kompiuteryje, taigi norint gauti detalesnius rezultatus reiktų išskaidyti blokų grandinės narius į labiau nutolusias virtualias mašinas ar į pasirinktą debesų platformą.

4.3 Tyrimo išvados

Apibendrinant šio skyriaus atliktus tyrimus galima teigti, jog šiame darbe siūlomas autentifikacijos sprendimas sunaudoja apie 4 kartus arba 0.3 MB daugiau operatyviosios atminties bei pareikalauja apie 30% daugiau laiko arba 0.07 sekundės daiktų įrenginio kliento dalyje saugesniam pirmam autentifikacijos faktoriui lyginant su atviru ir nepakankamai saugiu sprendimu, o antrasis autentifikavimo modulis blokų grandinės dalyje papildomai prideda vidutiniškai 0.35 sekundės atliekant įrenginio globalaus identifikatoriaus paiešką globalioje blokų grandinės būsenoje.

Nors lyginamieji procentiniai skaičiai yra pakankamai dideli, reiktų nepamiršti, jog įdiegtas sprendimas ne tik padidintų autentifikacijos saugą pridėdamas antrą faktorių, tačiau kartu apsaugotų komunikaciją saugiu TLS ryšiu. Be to, lyginant su jau analizuotais eksperimentiniais metodais – šis sprendimas parodė net 40% geresnį rezultatą lyginant žinučių apdorojimo metu RAM sunaudojimą.

Išvados ir rezultatai

Daiktų internetas ir išmanieji įrenginiai jau dabar užima didelę dalį aplink supančių įrenginių, kurie surenka, kaupia bei dalinasi gaunama informacija, todėl tokių įrenginių apsauga tampa aukščiausio prioriteto uždaviniu, o dauguma autorių išskiria autentifikaciją kaip sunkiausią uždavinį norint užtikrinti saugią komunikaciją.

Šio darbo analizės dalyje ištyrus eksperimentinius metodus bei įvairią literatūrą išsiaiškintos pagrindinės daiktų interneto problemos: nešifruotas bendravimas tarp daiktų interneto įrenginių bei serverių, nepakankamai saugi dažniausiai vieno faktoriaus autentifikacija ir infrastruktūros pagrindinis serveris, kuris sukuria SPOF problemą. Ištirti eksperimentiniai metodai šias problemas sprendžia tik dalinai ar ne visas. Analizės dalyje taip pat išsiaiškinta, jog blokų grandinių technologija paremtų metodų jau yra sukurta ir atliekant įvairius tyrimus įsitikinta šios technologijos panaudojamumu užtikrinant daiktų įrenginių patikimą autentifikaciją, konfidencialumą bei decentralizuojant pagrindinį autentifikacijos šaltinį, tačiau pastebėta, jog viešuose blokų grandinių tinkluose sudėtingiau užtikrinti konfidencialumą, patikimumą bei greitaveiką. Apibendrinus gautus rezultatus iškeltas pagrindinis darbo tikslas - sukurti saugesnį autentifikavimo metodą, kuris būtų paremtas privačia blokų grandinės technologija, o tikslui pasiekti apibrėžti konkretūs uždaviniai: pasiūlyti saugesnį DI įrenginių autentifikavimo metodą paremtą blokų grandinių technologiją bei pridedant antrą autentifikavimo faktorių, suprojektuoti siūlomą autentifikacijos platformą, sukurti pasiūlyto autentifikavimo metodo prototipą bei pateikti sukurto sprendimo tyrimo įvertinimą.

Pasiūlytas sprendimas išsprendžia iškeltas autentifikacijos problemas šiais komponentais: 2 žingsnių autentifikacija, daiktų įrenginių komunikacijos lengvu ir saugiu AMQPS protokolu bei privačios blokų grandinės platformos „Hyperledger Sawtooth“ panaudojimu galutinei autentifikacijai. Naudojant siūlomą metodą pirmasis autentifikacijos žingsnis įvykdomas patikrinant siuntėjo TLS sertifikatą bei klientą, kuriam sertifikatas išduotas žinučių brokerio „RabbitMQ“ registruotų klientų sąraše. Sėkmingai tenkinus pirmą sąlygą - žinutė ištransliuojama į blokų grandinės klientą, kuris atlieka antrąjį autentifikavimo žingsnį - patvirtina įrenginio identifikatorių blokų grandinės globalioje būsenoje ir tik sėkmingai atlikus abu žingsnius autentifikacija taip pat patvirtinama. Analizės dalyje taip pat išsiaiškinta, jog AMQP yra labiausiai tinkamas naudoti išmaniųjų įrenginių komunikacijai dėl mažų resursų reikalavimų bei galimybės enkapsuliuoti žinutę TLS protokolu, taip užšifruojant žinutę kriptografiškai ir užtikrinant saugų duomenų pernešimą.

Remiantis pasiūlytu metodu sėkmingai sukurtas prototipas, įrodantis suprojektuoto autentifikacijos modulio galimybes bei išskirtas būtinas saugumo charakteristikas. Atlikus sprendimo tyrimą panaudojant sukurtą prototipą išsiaiškinta, jog saugus ryšys pareikalauja apie 0.07 sekundės arba 30% daugiau laiko suformuoti žinutę daiktų interneto įrenginiui bei 0.3 MB arba 4 kartus daugiau operatyviosios atminties sąnaudų, o antras autentifikavimo faktorius prideda vidutiniškai apie 0.35 sekundes autentifikacijai atlikti. Tačiau palyginus kiekybinio tyrimo rezultatus su ištirtais eksperimentiniais metodais įsitikinta, jog šiame darbe pasiūlytas sprendimas pareikalauja apie 40% mažiau operatyviosios atminties sąnaudų atlikti autentifikaciją.

Apibendrinant atliktą darbą galima teigti išvadą, jog darbo uždaviniai yra atlikti ir pagrindinis darbo tikslas pasiektas - pasiūlytas ir sukurtas saugesnis daiktų interneto įrenginių autentifikavimo metodas paremtas blokų grandinės technologija.

Ateities tyrimų planas

Pagrindinis darbo tikslas bei uždaviniai yra įgyvendinti ir pasiūlytas saugesnis sprendimas bei sukurtas pirminis prototipas, tačiau būtina įvardinti ir ateities tyrimų planą, kuris apibrėžtų pagrindines gaires norint šį darbą tobulinti ir galimai paruošti naudojimui realiose sistemose:

- automatizuoti įrenginių registraciją su sertifikatų išdavimu, kadangi šiuo metu registracija reikalauja viską atlikti atskirais žingsniais ir administratorius yra atsakingas už šių žingsnių įgyvendinimą, sertifikatų įkėlimą į įrenginį kartu su kliento programėle bei įrenginio registraciją priskirtame „RabbitMQ“ serveryje;
- atlikti analizę su realiais daiktų interneto įrenginiais – taip būtų galima išsiaiškinti kaip reiktų modifikuoti kliento programą bei įrenginio registraciją siekiant palaikyti kiek įmanoma daugiau išmaniųjų įrenginių;
- išskaidyti blokų grandinės klientus geografiškai toliau nutolusiuose serveriuose ar virtualiose mašinose. Tai parodys realesnius kiekybinio tyrimo rezultatus, kadangi šio darbo kontekste tinklo pralaidumas bei pilna trukmė, reikalinga informacijai nukeliauti nuo daiktų įrenginio ir grįžti po sėkmingos autentifikacijos, nebuvo tiriami;
- patobulinti blokų grandinės tranzakcijų valdytoją sukuriant daugiau įrenginio valdymo funkcijų, pavyzdžiui, atnaujinti įrenginio informaciją;
- apgalvoti kur galimai turėtų būti kaupiami žurnalizavimo įrašai bei kaip surenkami iš skirtingų sistemos vietų taip padidinant infrastruktūros audito galimybes;
- atlikti kiekybinį tyrimą lyginant su realiais sprendimais egzistuojančiais rinkoje, tai padėtų išsiaiškinti bei išskirti stipriasias metodo savybes lyginant su populiariais sprendimais rinkoje.

Literatūra

[1] What's The Difference Between ZigBee And Z-Wave? [Tikrinta 2019-01-08]
http://www.electronicdesign.com/communications/what-s-difference-between-zigbee-and-z-wave

[2] Overview for 6LowPAN [Tikrinta 2019-01-08]
http://www.ti.com/wireless-connectivity/6lowpan/overview.html

[3] Thread standartas [Tikrinta 2019-01-08]
https://www.threadgroup.org/

[4] C. Occhiuzzi, S. Amendola, S. Manzari, S. Caizzone, G. Marrocco: Configurable radiofrequency identification sensing breadboard for industrial Internet of Things, Italy 2017 [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/document/7843807

[5] Daiktų interneto technologijos taikymo versle nauda ir rizika. 2015. Laima Zalieckaite, Raimundas Žilinskas. [Tikrinta 2019-01-01]
http://www.zurnalai.vu.lt/informacijos-mokslai/article/view/9223/7591

[6] Simple explanation of the Internet of Things [Tikrinta 2019-01-08]
https://www.forbes.com/sites/jacobmorgan/2014/05/13/simple-explanation-internet-things-that-anyon
#1c725ed21d09

[7] IoT connected devices worldwide [Tikrinta 2020-05-07]
https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/

[8] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M. and Ayyash, M. (2018). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications - IEEE Journals and Magazine. [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/abstract/document/7123563

[9] Qi Jing, Athanasios V. Vasilakos, Jiafu Wan, Jingwei LuDechao Qiu. Security in the internet of things: Perspectives and Challenged. In: Computer Science and Electronics Engineering (ICCSEE), 2012 international conference on. IEEE, 2012. p. 648-651 [Tikrinta 2020-05-24]
https://link.springer.com/article/10.1007/s11276-014-0761-7

[10] KARAGIANNIS, Vasileios, et al. A survey on application layer protocols for the internet of things. Transaction on IoT and Cloud Computing, 2015, 3.1: 11-17 [Tikrinta 2020-05-24]
https://www.researchgate.net/profile/Periklis_Chatzimisios/publication/303192188_A_survey_on_application_layer_protocols_for_the_Internet_of_Things/links/577b656608ae213761c9d91d.pdf

[11] Comparison of different message brokers at better.com [Tikrinta 2020-05-24]
https://better.engineering/message-queues/

[12] Pallavi Sethi and Smruti R. Sarangi, "Internet of Things: Architectures, Protocols, and Applications," Journal of Electrical and Computer Engineering, vol. 2017, Article ID 9324035, 25

pages, 2017 [Tikrinta 2020-05-24]
https://www.hindawi.com/journals/jece/2017/9324035/abs/

[13] LI, Xiong, et al. A robust ECC-based provable secure authentication protocol with privacy preserving for industrial internet of things. IEEE Transactions on Industrial Informatics, 2018, 14.8: 3599-3609 [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/abstract/document/8110708/

[14] MICHAEL, J. W.; COHN, ALAN; BUTCHER, JARED R. BlockChain technology. The Journal, 2018. [Tikrinta 2020-05-24]
https://www.steptoe.com/images/content/1/7/v2/171967/LIT-FebMar18-Feature-Blockchain.pdf

[15] MALIK, Nisha, et al. Blockchain Based Secured Identity Authentication and Expeditious Revocation Framework for Vehicular Networks. In: 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). IEEE, 2018. p. 674-679 [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/abstract/document/8455967

[16] HUH, Seyoung; CHO, Sangrae; KIM, Soohyung. Managing IoT devices using blockchain platform. In: Advanced Communication Technology (ICACT), 2017 19th International Conference on. IEEE, 2017. p. 464-467 [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/abstract/document/7890132

[17] RADZEVIČIUS, Žilvinas. Blokų Grandinėmis Grįstas Galinių įrenginių Autentifikavimo ūko Kompiuterijoje Metodas: Magistro Darbas. Kaunas: Kauno Technologijos Universitetas. Prieiga per ELABa – Nacionalinė Lietuvos Akademinė Elektroninė Biblioteka, 2019. [Tikrinta 2020-05-24]
https://epubl.ktu.edu/object/elaba:37396618/MAIN

[18] TESLYA, Nikolay; RYABCHIKOV, Igor. Blockchain Platforms Overview for Industrial IoT Purposes. In: 2018 22nd Conference of Open Innovations Association (FRUCT). IEEE, 2018. p. 250-256 [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/abstract/document/8468276

[19] WANG, Shuai, et al. An overview of smart contract: architecture, applications, and future trends. In: 2018 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2018. p. 108-113 [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/abstract/document/8500488

[20] SINGH, Jatinder, et al. Twenty security considerations for cloud-supported Internet of Things. IEEE Internet of things Journal, 2016, 3.3: 269-284. [Tikrinta 2020-05-24]
https://ieeexplore.ieee.org/document/7165580

[21] MQTT komunikacijos protokolas [Tikrinta 2019-03-04]
http://mqtt.org/

[22] Lucy Zhang. Building Facebook Messenger. 2011. [Tikrinta 2020-05-24]
https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920

[23] AMQP komunikacijos protokolas [Tikrinta 2019-03-04]
https://www.amqp.org

[24] FERNANDES, Joel L., et al. Performance evaluation of RESTful web services and AMQP protocol. In: Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on. IEEE, 2013. p. 810-815. [Tikrinta 2020-01-19]
https://ieeexplore.ieee.org/abstract/document/6614932

[25] Solidity - high-level language for writing smart contracts [Tikrinta 2020-01-19]
https://solidity.readthedocs.io

[26] NAIK, Nitin. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: 2017 IEEE international systems engineering symposium (ISSE). IEEE, 2017. p. 1-7. [Tikrinta 2020-01-19]
https://ieeexplore.ieee.org/abstract/document/8088251

[27] SANTOSO, Freddy K.; VUN, Nicholas CH. Securing IoT for smart home system. In: Consumer Electronics (ISCE), 2015 IEEE International Symposium on. IEEE, 2015. p. 1-2. [Tikrinta 2020-01-19]
https://ieeexplore.ieee.org/abstract/document/7177843

[28] LUZURIAGA, Jorge E., et al. A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In: 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC). IEEE, 2015. p. 931-936. [Tikrinta 2020-01-19]
https://ieeexplore.ieee.org/abstract/document/7158101

[29] AMQP 1.0 core security [Tikrinta 2020-01-19]
http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-security-v1.0.html

[30] Docker containerization platform [Tikrinta 2020-01-19]
https://www.docker.com/

[31] LAL, Nilesh A.; PRASAD, Salendra; FARIK, Mohammed. A review of authentication methods. Int. J. Sci. Technology Res, 2016, 5.11: 246-249. [Tikrinta 2020-01-19]
https://pdfs.semanticscholar.org/7977/08bd4f854a125ec20865d4b7be76a1aa4740.pdf

[32] Eric Rescorla 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. [Tikrinta 2020-01-19]
https://tools.ietf.org/html/rfc8446

[33] Proposed safe machine ID usage [Tikrinta 2020-01-19]
http://manpages.ubuntu.com/manpages/bionic/man3/sd_id128_get_machine_app_specific.3.html

[34] „tls-gen“ įrankio šaltinio kodas [Tikrinta 2020-05-24]
https://github.com/michaelklishin/tls-gen

[35] - IBM pateikta AMQP žinutės struktūra [Tikrinta 2020-05-24]
https://www.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.dev.doc/q125040_.htm

[36] - „Hyperledger Sawtooth“ oficiali „Python“ kalbos biblioteka [Tikrinta 2020-05-24]
https://github.com/hyperledger/sawtooth-sdk-python

[37] - „Python“ kalbos biblioteka skirta stebėti resursų panaudojimą šios kalbos programų [Tikrinta 2020-05-24]
https://pypi.org/project/memory-profiler/

[38] - „Python“ kalbos biblioteka skirta išmatuoti mažų kodo fragmentų vykdymo laiką [Tikrinta 2020-05-24]
https://docs.python.org/3.8/library/timeit.html

Priedai

A Blokų grandinės tarpininko kliento scenarijaus kodas

6 išeities kodas. Tarpininko kliento programėlė (controller.py)

```
1 import json
2 import logging
3 import subprocess
4 import pika
5 import ssl
6 import sqlite3
7
8 def create_connection(db_file):
9     conn = None
10    try:
11        conn = sqlite3.connect(db_file)
12        return conn
13    except sqlite3.Error as e:
14        print(e)
15
16    return conn
17
18 sql_create_entries_table = """ CREATE TABLE IF NOT EXISTS device_data (
19                                     id integer PRIMARY KEY,
20                                     device_id text NOT NULL,
21                                     data text,
22                                     device_name text
23                                 ); """
24
25 def create_table(conn, create_table_sql):
26    try:
27        c = conn.cursor()
28        c.execute(create_table_sql)
29    except sqlite3.Error as e:
30        print(e)
31
32 def insert_data(conn, device_data):
33    sql = ''' INSERT INTO device_data(device_id,data,device_name)
34              VALUES(?,?,?) '''
35    cur = conn.cursor()
36    cur.execute(sql, device_data)
37    return cur.lastrowid
38
39
40 db_conn = create_connection(r"pythonsqlite.db")
41 if db_conn is not None:
42    create_table(db_conn, sql_create_entries_table)
43
44 logging.basicConfig(level=logging.INFO)
45 context = ssl.create_default_context(cafile="ca_certificate.pem")
```

```
46 context.load_cert_chain("client_certificate.pem", "client_key.pem")
47 ssl_options = pika.SSLOptions(context, "DESKTOP-O3O8HC4")
48 conn_params = pika.ConnectionParameters(host="host.docker.internal", port=5671, virtual_host="/",
       ssl_options=ssl_options, credentials=pika.credentials.ExternalCredentials())
49
50 connection = pika.BlockingConnection(conn_params)
51 channel = connection.channel()
52 channel.queue_declare("master")
53
54 def callback(ch, method, properties, body):
55     process = subprocess.run(['control', 'show', '--url', 'http://sawtooth-rest-api-default-0:8008',
           properties.app_id], stdout=subprocess.PIPE, universal_newlines=True)
56     try:
57         data = json.loads(process.stdout)
58     except json.JSONDecodeError as e:
59         data = None
60
61     if data and data['is_active'] == "true":
62         insert_data(db_conn, (data['device_id'], body.decode('utf-8'), data['name']))
63
64 channel.basic_consume(queue="master", on_message_callback=callback, auto_ack=True)
65 channel.start_consuming()
```

B Blokų grandinės kliento programėlės komandinės eilutės kodas

7 išeities kodas. Tarpininko kliento programėlės CLI kodas (control_cli.py)

```
 1 from __future__ import print_function
 2
 3 import argparse
 4 import getpass
 5 import json
 6 import logging
 7 import os
 8 import traceback
 9 import sys
10 import pkg_resources
11
12 from colorlog import ColoredFormatter
13
14 from sawtooth_control.control_client import ControlClient
15 from sawtooth_control.control_exceptions import ControlException
16
17 DISTRIBUTION_NAME = 'sawtooth-control'
18
19 DEFAULT_URL = 'http://127.0.0.1:8008'
20
21 def create_console_handler(verbose_level):
22     clog = logging.StreamHandler()
```

```
23      formatter = ColoredFormatter(
24          "%(log_color)s[%(asctime)s %(levelname)-8s%(module)s]%(reset)s "
25          "%(white)s%(message)s",
26          datefmt="%H:%M:%S",
27          reset=True,
28          log_colors={
29              'DEBUG': 'cyan',
30              'INFO': 'green',
31              'WARNING': 'yellow',
32              'ERROR': 'red',
33              'CRITICAL': 'red',
34          })
35
36      clog.setFormatter(formatter)
37
38      if verbose_level == 0:
39          clog.setLevel(logging.WARN)
40      elif verbose_level == 1:
41          clog.setLevel(logging.INFO)
42      else:
43          clog.setLevel(logging.DEBUG)
44
45      return clog
46
47
48  def setup_loggers(verbose_level):
49      logger = logging.getLogger()
50      logger.setLevel(logging.DEBUG)
51      logger.addHandler(create_console_handler(verbose_level))
52
53
54  def add_create_parser(subparsers, parent_parser):
55      parser = subparsers.add_parser(
56          'create',
57          help='Creates a new device',
58          description='creates a new device while creating a transaction in the blockchain',
59          parents=[parent_parser])
60
61      parser.add_argument(
62          'name',
63          type=str,
64          help='readable identifier for the device')
65
66      parser.add_argument(
67          'device_id',
68          type=str,
69          help='unique identifier a.k.a. device id')
70
71      parser.add_argument(
72          'is_active',
73          type=str,
74          help='is active device')
75
```

```
76      parser.add_argument(
77          '--url',
78          type=str,
79          help='specify URL of REST API')
80
81      parser.add_argument(
82          '--key-dir',
83          type=str,
84          help="identify directory of user's private key file")
85
86      parser.add_argument(
87          '--disable-client-validation',
88          action='store_true',
89          default=False,
90          help='disable client validation')
91
92      parser.add_argument(
93          '--wait',
94          nargs='?',
95          const=sys.maxsize,
96          type=int,
97          help='set time, in seconds, to wait for device to commit')
98
99
100 def add_list_parser(subparsers, parent_parser):
101     parser = subparsers.add_parser(
102         'list',
103         help='Displays information for all devices',
104         description='Displays information for all devices in state',
105         parents=[parent_parser])
106
107     parser.add_argument(
108         '--url',
109         type=str,
110         help='specify URL of REST API')
111
112
113 def add_show_parser(subparsers, parent_parser):
114     parser = subparsers.add_parser(
115         'show',
116         help='Displays information about a device',
117         description='Displays the complete device info',
118         parents=[parent_parser])
119
120     parser.add_argument(
121         'device_id',
122         type=str,
123         help='unique identifier a.k.a. device id')
124
125     parser.add_argument(
126         '--url',
127         type=str,
128         help='specify URL of REST API')
```

```
129
130
131 def add_delete_parser(subparsers, parent_parser):
132     parser = subparsers.add_parser('delete', parents=[parent_parser])
133
134     parser.add_argument(
135         'device_id',
136         type=str,
137         help='unique identifier a.k.a. device id')
138
139     parser.add_argument(
140         '--url',
141         type=str,
142         help='specify URL of REST API')
143
144     parser.add_argument(
145         '--wait',
146         nargs='?',
147         const=sys.maxsize,
148         type=int,
149         help='set time, in seconds, to wait for delete transaction to commit')
150
151
152 def create_parent_parser(prog_name):
153     parent_parser = argparse.ArgumentParser(prog=prog_name, add_help=False)
154     parent_parser.add_argument(
155         '-v', '--verbose',
156         action='count',
157         help='enable more verbose output')
158
159     try:
160         version = pkg_resources.get_distribution(DISTRIBUTION_NAME).version
161     except pkg_resources.DistributionNotFound:
162         version = 'UNKNOWN'
163
164     parent_parser.add_argument(
165         '-V', '--version',
166         action='version',
167         version=(DISTRIBUTION_NAME + ' (Hyperledger Sawtooth) version {}')
168         .format(version),
169         help='display version information')
170
171     return parent_parser
172
173
174 def create_parser(prog_name):
175     parent_parser = create_parent_parser(prog_name)
176
177     parser = argparse.ArgumentParser(
178         description='Provides commands to control IoT authenticator network',
179         parents=[parent_parser])
180
181     subparsers = parser.add_subparsers(title='subcommands', dest='command')
```

```
182
183         subparsers.required = True
184
185         add_create_parser(subparsers, parent_parser)
186         add_list_parser(subparsers, parent_parser)
187         add_show_parser(subparsers, parent_parser)
188         add_delete_parser(subparsers, parent_parser)
189
190         return parser
191
192
193 def do_list(args):
194         url = _get_url(args)
195
196         client = ControlClient(base_url=url, keyfile=None)
197
198         devices_list = [
199             {"name": device.split(',')[0], "device_id": device.split(',')[1], "is_active": device.split(',')[2]}
200             for devices in client.list()
201             for device in devices.decode().split('l')
202         ]
203         print(json.dumps(devices_list))
204
205
206 def do_show(args):
207         device_id = args.device_id
208
209         url = _get_url(args)
210
211         client = ControlClient(base_url=url, keyfile=None)
212
213         data = client.show(device_id)
214         if data.decode():
215             name, device_id, is_active = data.decode().split('l')[0].split(',')
216             print(json.dumps({"name": name, "device_id": device_id, "is_active": is_active}))
217         else:
218             print(json.dumps({}))
219
220
221
222 def do_create(args):
223         name = args.name
224         device_id = args.device_id
225         pub_key = args.is_active
226
227         url = _get_url(args)
228         keyfile = _get_keyfile(args)
229
230         client = ControlClient(base_url=url, keyfile=keyfile)
231
232         if args.wait and args.wait > 0:
233             response = client.create(name, device_id, pub_key, wait=args.wait)
234         else:
```

```
235             response = client.create(name, device_id, pub_key)
236
237         print("Response: {}".format(response))
238
239
240
241 def do_delete(args):
242         device_id = args.device_id
243
244         url = _get_url(args)
245         keyfile = _get_keyfile(args)
246
247         client = ControlClient(base_url=url, keyfile=keyfile)
248
249         if args.wait and args.wait > 0:
250             response = client.delete(device_id, wait=args.wait)
251         else:
252             response = client.delete(device_id)
253
254         print("Response: {}".format(response))
255
256
257 def _get_url(args):
258         return DEFAULT_URL if args.url is None else args.url
259
260
261 def _get_keyfile(args):
262         username = getpass.getuser()
263         home = os.path.expanduser("~")
264         key_dir = os.path.join(home, ".sawtooth", "keys")
265
266         return '{}/{}.priv'.format(key_dir, username)
267
268
269 def _get_auth_info(args):
270         auth_user = args.auth_user
271         auth_password = args.auth_password
272         if auth_user is not None and auth_password is None:
273             auth_password = getpass.getpass(prompt="Auth Password: ")
274
275         return auth_user, auth_password
276
277
278 def main(prog_name=os.path.basename(sys.argv[0]), args=None):
279         if args is None:
280             args = sys.argv[1:]
281         parser = create_parser(prog_name)
282         args = parser.parse_args(args)
283
284         if args.verbose is None:
285             verbose_level = 0
286         else:
287             verbose_level = args.verbose
```

```
288
289     setup_loggers(verbose_level=verbose_level)
290
291     if args.command == 'create':
292         do_create(args)
293     elif args.command == 'list':
294         do_list(args)
295     elif args.command == 'show':
296         do_show(args)
297     elif args.command == 'delete':
298         do_delete(args)
299     else:
300         raise ControlException("invalid command: {}".format(args.command))
301
302
303 def main_wrapper():
304     try:
305         main()
306     except ControlException as err:
307         print("Error: {}".format(err), file=sys.stderr)
308         sys.exit(1)
309     except KeyboardInterrupt:
310         pass
311     except SystemExit as err:
312         raise err
313     except BaseException as err:
314         traceback.print_exc(file=sys.stderr)
315         sys.exit(1)
```

C Blokų grandinės kliento programėlės kliento kodas

8 išeities kodas. Tarpininko kliento programėlės kliento kodas (control_client.py)

```
 1 import hashlib
 2 import base64
 3 import time
 4 import random
 5 import requests
 6 import yaml
 7
 8 from sawtooth_signing import create_context
 9 from sawtooth_signing import CryptoFactory
10 from sawtooth_signing import ParseError
11 from sawtooth_signing.secp256k1 import Secp256k1PrivateKey
12
13 from sawtooth_sdk.protobuf.transaction_pb2 import TransactionHeader
14 from sawtooth_sdk.protobuf.transaction_pb2 import Transaction
15 from sawtooth_sdk.protobuf.batch_pb2 import BatchList
16 from sawtooth_sdk.protobuf.batch_pb2 import BatchHeader
17 from sawtooth_sdk.protobuf.batch_pb2 import Batch
18
```

```
19 from sawtooth_control.control_exceptions import ControlException
20
21
22 def _sha512(data):
23     return hashlib.sha512(data).hexdigest()
24
25
26 class ControlClient:
27     def __init__(self, base_url, keyfile=None):
28
29         self._base_url = base_url
30
31         if keyfile is not None:
32             try:
33                 with open(keyfile) as fd:
34                     private_key_str = fd.read().strip()
35                     fd.close()
36             except OSError as err:
37                 raise ControlException(
38                     'Failed to read private key: {}'.format(str(err)))
39
40             try:
41                 private_key = Secp256k1PrivateKey.from_hex(private_key_str)
42             except ParseError as e:
43                 raise ControlException(
44                     'Unable to load private key: {}'.format(str(e)))
45
46             self._signer = CryptoFactory(
47                 create_context('secp256k1')).new_signer(private_key)
48
49     def create(self, name, device_id, is_active, wait=None):
50         return self._send_device_txn(name, "create", device_id, is_active, wait=wait)
51
52     def delete(self, name, wait=None):
53         return self._send_device_txn(name, "delete", wait=wait,)
54
55     def list(self):
56         control_prefix = self._get_prefix()
57
58         result = self._send_request("state?address={}".format(control_prefix))
59
60         try:
61             encoded_entries = yaml.safe_load(result)["data"]
62
63             return [
64                 base64.b64decode(entry["data"]) for entry in encoded_entries
65             ]
66
67         except BaseException:
68             return None
69
70     def show(self, name):
71         address = self._get_address(name)
```

```
 72
 73          result = self._send_request("state/{}".format(address), name=name)
 74          try:
 75              return base64.b64decode(yaml.safe_load(result)["data"])
 76
 77          except BaseException:
 78              return None
 79
 80      def _get_status(self, batch_id, wait):
 81          try:
 82              result = self._send_request('batch_statuses?id={}&wait={}'.format(batch_id, wait))
 83              return yaml.safe_load(result)['data'][0]['status']
 84          except BaseException as err:
 85              raise ControlException(err)
 86
 87      def _get_prefix(self):
 88          return _sha512('control'.encode('utf-8'))[0:6]
 89
 90      def _get_address(self, name):
 91          control_prefix = self._get_prefix()
 92          device_address = _sha512(name.encode('utf-8'))[0:64]
 93          return control_prefix + device_address
 94
 95      def _send_request(self,
 96                        suffix,
 97                        data=None,
 98                        content_type=None,
 99                        name=None):
100          if self._base_url.startswith("http://"):
101              url = "{}/{}".format(self._base_url, suffix)
102          else:
103              url = "http://{}/{}".format(self._base_url, suffix)
104
105          headers = {}
106
107          if content_type is not None:
108              headers['Content-Type'] = content_type
109
110          try:
111              if data is not None:
112                  result = requests.post(url, headers=headers, data=data)
113              else:
114                  result = requests.get(url, headers=headers)
115
116              if result.status_code == 404:
117                  raise ControlException("No such device: {}".format(name))
118
119              if not result.ok:
120                  raise ControlException("Error {}: {}".format(
121                      result.status_code, result.reason))
122
123          except requests.ConnectionError as err:
124              raise ControlException(
```

```
125                 'Failed to connect to {}: {}'.format(url, str(err)))
126
127         except BaseException as err:
128             raise ControlException(err)
129
130         return result.text
131
132     def _send_device_txn(self, name, action, device_id="", is_active="", wait=None):
133         payload = ",".join([name, action, device_id, is_active]).encode()
134
135         # Construct the address
136         address = self._get_address(device_id)
137
138         header = TransactionHeader(
139             signer_public_key=self._signer.get_public_key().as_hex(),
140             family_name="control",
141             family_version="1.0",
142             inputs=[address],
143             outputs=[address],
144             dependencies=[],
145             payload_sha512=_sha512(payload),
146             batcher_public_key=self._signer.get_public_key().as_hex(),
147             nonce=hex(random.randint(0, 2**64))
148         ).SerializeToString()
149
150         signature = self._signer.sign(header)
151
152         transaction = Transaction(
153             header=header,
154             payload=payload,
155             header_signature=signature
156         )
157
158         batch_list = self._create_batch_list([transaction])
159         batch_id = batch_list.batches[0].header_signature
160
161         if wait and wait > 0:
162             wait_time = 0
163             start_time = time.time()
164             response = self._send_request(
165                 "batches", batch_list.SerializeToString(),
166                 'application/octet-stream')
167             while wait_time < wait:
168                 status = self._get_status(
169                     batch_id,
170                     wait - int(wait_time))
171                 wait_time = time.time() - start_time
172
173                 if status != 'PENDING':
174                     return response
175
176             return response
177
```

```
178             return self._send_request(
179                 "batches", batch_list.SerializeToString(),
180                 'application/octet-stream')
181
182     def _create_batch_list(self, transactions):
183         transaction_signatures = [t.header_signature for t in transactions]
184
185         header = BatchHeader(
186             signer_public_key=self._signer.get_public_key().as_hex(),
187             transaction_ids=transaction_signatures
188         ).SerializeToString()
189
190         signature = self._signer.sign(header)
191
192         batch = Batch(
193             header=header,
194             transactions=transactions,
195             header_signature=signature)
196         return BatchList(batches=[batch])
```

D Blokų grandinės tranzakcijos valdytojo turinio kodas

9 išeities kodas. Blokų grandinės tranzakcijos valdytojo turinio kodas (control_payload.py)

```
1 from sawtooth_sdk.processor.exceptions import InvalidTransaction
2
3
4 class ControlPayload:
5
6     def __init__(self, payload):
7         try:
8             # The payload is csv utf-8 encoded string
9             name, action, device_id, is_active = payload.decode().split(",")
10        except ValueError:
11            raise InvalidTransaction("Invalid payload serialization")
12
13        if not action:
14            raise InvalidTransaction('Action is required')
15
16        if action not in ('create', 'delete'):
17            raise InvalidTransaction('Invalid action: {}'.format(action))
18
19        if action == 'create':
20            if not name or not is_active or not device_id:
21                raise InvalidTransaction('Name, active tag and device id are required')
22
23        self._name = name
24        self._action = action
25        self._device_id = device_id
26        self._is_active = is_active
27
```

```
28      @staticmethod
29      def from_bytes(payload):
30          return ControlPayload(payload=payload)
31
32      @property
33      def name(self):
34          return self._name
35
36      @property
37      def device_id(self):
38          return self._device_id
39
40      @property
41      def is_active(self):
42          return self._is_active
43
44      @property
45      def action(self):
46          return self._action
```

E Blokų grandinės tranzakcijos valdytojo būsenos kodas

10 išeities kodas. Blokų grandinės tranzakcijos valdytojo būsenos kodas (control_state.py)

```
 1 import hashlib
 2
 3 from sawtooth_sdk.processor.exceptions import InternalError
 4
 5
 6 CONTROL_NAMESPACE = hashlib.sha512('control'.encode("utf−8")).hexdigest()[0:6]
 7
 8
 9 def _make_control_address(device_id):
10     return CONTROL_NAMESPACE + hashlib.sha512(device_id.encode('utf−8')).hexdigest()[:64]
11
12
13 class SmartDevice:
14     def __init__(self, name, device_id="", is_active=""):
15         self.name = name
16         self.device_id = device_id
17         self.is_active = is_active
18
19
20 class ControlState:
21
22     TIMEOUT = 3
23
24     def __init__(self, context):
25         self._context = context
26         self._address_cache = {}
27
```

```
    def delete_device(self, device_id):
        devices = self._load_devices(device_id=device_id)

        del devices[device_id]
        if devices:
            self._store_device(device_id, devices=devices)
        else:
            self._delete_device(device_id)

    def set_device(self, device_id, device):
                devices = self._load_devices(device_id=device_id)

        devices[device_id] = device

        self._store_device(device_id, devices=devices)

    def get_device(self, device_id):
                return self._load_devices(device_id=device_id).get(device_id)

    def _store_device(self, device_id, devices):
        address = _make_control_address(device_id)

        state_data = self._serialize(devices)

        self._address_cache[address] = state_data

        self._context.set_state(
            {address: state_data},
            timeout=self.TIMEOUT)

    def _delete_device(self, device_id):
        address = _make_control_address(device_id)

        self._context.delete_state(
            [address],
            timeout=self.TIMEOUT)

        self._address_cache[address] = None

    def _load_devices(self, device_id):
        address = _make_control_address(device_id)

        if address in self._address_cache:
            if self._address_cache[address]:
                serialized_games = self._address_cache[address]
                games = self._deserialize(serialized_games)
            else:
                games = {}
        else:
            state_entries = self._context.get_state(
                [address],
                timeout=self.TIMEOUT)
            if state_entries:
```

```
81
82                      self._address_cache[address] = state_entries[0].data
83
84                      games = self._deserialize(data=state_entries[0].data)
85
86                  else:
87                      self._address_cache[address] = None
88                      games = {}
89
90          return games
91
92      def _deserialize(self, data):
93                      games = {}
94          try:
95              for game in data.decode().split("|"):
96                  name, device_id, pub_key = game.split(",")
97
98                  games[device_id] = SmartDevice(name, device_id, pub_key)
99          except ValueError:
100             raise InternalError("Failed to deserialize device data")
101
102         return games
103
104     def _serialize(self, devices):
105         device_strs = []
106         for device_id, g in devices.items():
107             device_str = ",".join(
108                 [g.name, device_id, g.is_active])
109             device_strs.append(device_str)
110
111         return "|".join(sorted(device_strs)).encode()
```

F Blokų grandinės tranzakcijos valdytojo klasės kodas

11 išeities kodas. Blokų grandinės tranzakcijos valdytojo klasės kodas (handler.py)

```
1 import logging
2
3
4 from sawtooth_sdk.processor.handler import TransactionHandler
5 from sawtooth_sdk.processor.exceptions import InvalidTransaction
6
7 from sawtooth_control.processor.control_payload import ControlPayload
8 from sawtooth_control.processor.control_state import SmartDevice
9 from sawtooth_control.processor.control_state import ControlState
10 from sawtooth_control.processor.control_state import CONTROL_NAMESPACE
11
12
13 LOGGER = logging.getLogger(__name__)
14
15
```

```
16 class ControlTransactionHandler(TransactionHandler):
17     @property
18     def family_name(self):
19         return 'control'
20
21     @property
22     def family_versions(self):
23         return ['1.0']
24
25     @property
26     def namespaces(self):
27         return [CONTROL_NAMESPACE]
28
29     def apply(self, transaction, context):
30         control_payload = ControlPayload.from_bytes(transaction.payload)
31
32         control_state = ControlState(context)
33
34         if control_payload.action == 'delete':
35             device = control_state.get_device(control_payload.device_id)
36
37             if device is None:
38                 raise InvalidTransaction('Invalid action: device does not exist')
39
40             control_state.delete_device(control_payload.name)
41
42         elif control_payload.action == 'create':
43             device = control_state.get_device(control_payload.device_id)
44             if device is not None:
45                 raise InvalidTransaction('Invalid action: Device already exists: {}'.format(
                        control_payload.device_id))
46
47             game = SmartDevice(name=control_payload.name,
48                                device_id=control_payload.device_id,
49                                is_active=control_payload.is_active)
50
51             control_state.set_device(control_payload.device_id, game)
52
53         else:
54             raise InvalidTransaction('Unhandled action: {}'.format(control_payload.action))
```

G Blokų grandinės tranzakcijos valdytojo pagrindinis kodas

12 išeities kodas. Blokų grandinės tranzakcijos valdytojo pagrindinis kodas (main.py)

```
1 import sys
2 import os
3 import argparse
4 import pkg_resources
5
6 from sawtooth_sdk.processor.core import TransactionProcessor
```

```
 7 from sawtooth_sdk.processor.log import init_console_logging
 8 from sawtooth_sdk.processor.log import log_configuration
 9 from sawtooth_sdk.processor.config import get_log_config
10 from sawtooth_sdk.processor.config import get_log_dir
11 from sawtooth_sdk.processor.config import get_config_dir
12 from sawtooth_control.processor.handler import ControlTransactionHandler
13 from sawtooth_control.processor.config.control import XOConfig
14 from sawtooth_control.processor.config.control import load_default_xo_config
15 from sawtooth_control.processor.config.control import load_toml_xo_config
16 from sawtooth_control.processor.config.control import merge_xo_config
17
18
19 DISTRIBUTION_NAME = 'sawtooth-control'
20
21
22 def parse_args(args):
23     parser = argparse.ArgumentParser(
24         formatter_class=argparse.RawTextHelpFormatter)
25
26     parser.add_argument(
27         '-C', '--connect',
28         help='Endpoint for the validator connection')
29
30     parser.add_argument('-v', '--verbose',
31                         action='count',
32                         default=0,
33                         help='Increase output sent to stderr')
34
35     try:
36         version = pkg_resources.get_distribution(DISTRIBUTION_NAME).version
37     except pkg_resources.DistributionNotFound:
38         version = 'UNKNOWN'
39
40     parser.add_argument(
41         '-V', '--version',
42         action='version',
43         version=(DISTRIBUTION_NAME + ' (Hyperledger Sawtooth) version {}')
44         .format(version),
45         help='print version information')
46
47     return parser.parse_args(args)
48
49
50 def load_xo_config(first_config):
51     default_xo_config = \
52         load_default_xo_config()
53     conf_file = os.path.join(get_config_dir(), 'control.toml')
54
55     toml_config = load_toml_xo_config(conf_file)
56
57     return merge_xo_config(
58         configs=[first_config, toml_config, default_xo_config])
59
```

```
60
61 def create_xo_config(args):
62     return XOConfig(connect=args.connect)
63
64
65 def main(args=None):
66     if args is None:
67         args = sys.argv[1:]
68     opts = parse_args(args)
69     processor = None
70     try:
71         arg_config = create_xo_config(opts)
72         xo_config = load_xo_config(arg_config)
73         processor = TransactionProcessor(url=xo_config.connect)
74         log_config = get_log_config(filename="control_log_config.toml")
75
76         # If no toml, try loading yaml
77         if log_config is None:
78             log_config = get_log_config(filename="control_log_config.yaml")
79
80         if log_config is not None:
81             log_configuration(log_config=log_config)
82         else:
83             log_dir = get_log_dir()
84             # use the transaction processor zmq identity for filename
85             log_configuration(
86                 log_dir=log_dir,
87                 name="control-" + str(processor.zmq_id)[2:-1])
88
89         init_console_logging(verbose_level=opts.verbose)
90
91         handler = ControlTransactionHandler()
92
93         processor.add_handler(handler)
94
95         processor.start()
96     except KeyboardInterrupt:
97         pass
98     except Exception as e: # pylint: disable=broad-except
99         print("Error: {}".format(e))
100    finally:
101        if processor is not None:
102            processor.stop()
```

H Sprendimo infrastruktūros „Docker Compose“ konfigūracijos failas

13 išeities kodas. „Docker Compose“ konfigūracijos failas (sawtooth-default-poet.yaml)

```
1 version: "2.1"
```

```
 2
 3 volumes:
 4   poet-shared:
 5
 6 services:
 7   shell:
 8     image: sawtooth-shell-default:latest
 9     container_name: sawtooth-shell-default
10     volumes:
11         - ./sawtooth-sdk-python:/root
12     entrypoint: "bash -c \"\
13         sawtooth keygen && \
14         tail -f /dev/null \
15         \""
16
17   validator-0:
18     image: hyperledger/sawtooth-validator:chime
19     container_name: sawtooth-validator-default-0
20     expose:
21       - 4004
22       - 5050
23       - 8800
24     volumes:
25       - poet-shared:/poet-shared
26     command: "bash -c \"\
27         sawadm keygen --force && \
28         mkdir -p /poet-shared/validator-0 || true && \
29         cp -a /etc/sawtooth/keys /poet-shared/validator-0/ && \
30         while [ ! -f /poet-shared/poet-enclave-measurement ]; do sleep 1; done && \
31         while [ ! -f /poet-shared/poet-enclave-basename ]; do sleep 1; done && \
32         while [ ! -f /poet-shared/poet.batch ]; do sleep 1; done && \
33         cp /poet-shared/poet.batch / && \
34         sawset genesis \
35           -k /etc/sawtooth/keys/validator.priv \
36           -o config-genesis.batch && \
37         sawset proposal create \
38           -k /etc/sawtooth/keys/validator.priv \
39           sawtooth.consensus.algorithm.name=PoET \
40           sawtooth.consensus.algorithm.version=0.1 \
41           sawtooth.poet.report_public_key_pem=\
42           \\\"$$(cat /poet-shared/simulator_rk_pub.pem)\\\" \
43           sawtooth.poet.valid_enclave_measurements=$$(cat /poet-shared/poet-enclave-measurement)
                 \
44           sawtooth.poet.valid_enclave_basenames=$$(cat /poet-shared/poet-enclave-basename) \
45           -o config.batch && \
46         sawset proposal create \
47           -k /etc/sawtooth/keys/validator.priv \
48             sawtooth.poet.target_wait_time=5 \
49             sawtooth.poet.initial_wait_time=25 \
50             sawtooth.publisher.max_batches_per_block=100 \
51           -o poet-settings.batch && \
52         sawadm genesis \
53           config-genesis.batch config.batch poet.batch poet-settings.batch && \
```

```
54            sawtooth-validator -v \
55              --bind network:tcp://eth0:8800 \
56              --bind component:tcp://eth0:4004 \
57              --bind consensus:tcp://eth0:5050 \
58              --peering static \
59              --endpoint tcp://validator-0:8800 \
60              --scheduler parallel \
61              --network-auth trust
62      \""
63      environment:
64        PYTHONPATH: "/project/sawtooth-core/consensus/poet/common:\
65          /project/sawtooth-core/consensus/poet/simulator:\
66          /project/sawtooth-core/consensus/poet/core"
67      stop_signal: SIGKILL
68
69    validator-1:
70      image: hyperledger/sawtooth-validator:chime
71      container_name: sawtooth-validator-default-1
72      expose:
73        - 4004
74        - 5050
75        - 8800
76      volumes:
77        - poet-shared:/poet-shared
78      command: |
79        bash -c "
80          sawadm keygen --force && \
81          mkdir -p /poet-shared/validator-1 || true && \
82          cp -a /etc/sawtooth/keys /poet-shared/validator-1/ && \
83          sawtooth-validator -v \
84                --bind network:tcp://eth0:8800 \
85                --bind component:tcp://eth0:4004 \
86                --bind consensus:tcp://eth0:5050 \
87                --peering static \
88                --endpoint tcp://validator-1:8800 \
89                --peers tcp://validator-0:8800 \
90                --scheduler parallel \
91                --network-auth trust
92        "
93      environment:
94        PYTHONPATH: "/project/sawtooth-core/consensus/poet/common:\
95          /project/sawtooth-core/consensus/poet/simulator:\
96          /project/sawtooth-core/consensus/poet/core"
97      stop_signal: SIGKILL
98
99    validator-2:
100     image: hyperledger/sawtooth-validator:chime
101     container_name: sawtooth-validator-default-2
102     expose:
103       - 4004
104       - 5050
105       - 8800
106     volumes:
```

```
107       - poet-shared:/poet-shared
108     command: |
109       bash -c "
110         sawadm keygen --force && \
111         mkdir -p /poet-shared/validator-2 && \
112         cp -a /etc/sawtooth/keys /poet-shared/validator-2/ && \
113         sawtooth-validator -v \
114               --bind network:tcp://eth0:8800 \
115               --bind component:tcp://eth0:4004 \
116               --bind consensus:tcp://eth0:5050 \
117               --peering static \
118               --endpoint tcp://validator-2:8800 \
119               --peers tcp://validator-0:8800,tcp://validator-1:8800 \
120               --scheduler parallel \
121               --network-auth trust
122       "
123     environment:
124       PYTHONPATH: "/project/sawtooth-core/consensus/poet/common:\
125         /project/sawtooth-core/consensus/poet/simulator:\
126         /project/sawtooth-core/consensus/poet/core"
127     stop_signal: SIGKILL
128
129   validator-3:
130     image: hyperledger/sawtooth-validator:chime
131     container_name: sawtooth-validator-default-3
132     expose:
133       - 4004
134       - 5050
135       - 8800
136     volumes:
137       - poet-shared:/poet-shared
138     command: |
139       bash -c "
140         sawadm keygen --force && \
141         mkdir -p /poet-shared/validator-3 && \
142         cp -a /etc/sawtooth/keys /poet-shared/validator-3/ && \
143         sawtooth-validator -v \
144               --bind network:tcp://eth0:8800 \
145               --bind component:tcp://eth0:4004 \
146               --bind consensus:tcp://eth0:5050 \
147               --peering static \
148               --endpoint tcp://validator-3:8800 \
149               --peers tcp://validator-0:8800,tcp://validator-1:8800,tcp://validator-2:8800 \
150               --scheduler parallel \
151               --network-auth trust
152       "
153     environment:
154       PYTHONPATH: "/project/sawtooth-core/consensus/poet/common:\
155         /project/sawtooth-core/consensus/poet/simulator:\
156         /project/sawtooth-core/consensus/poet/core"
157     stop_signal: SIGKILL
158
159   validator-4:
```

```
160     image: hyperledger/sawtooth-validator:chime
161     container_name: sawtooth-validator-default-4
162     expose:
163       - 4004
164       - 5050
165       - 8800
166     volumes:
167       - poet-shared:/poet-shared
168     command: |
169       bash -c "
170         sawadm keygen --force && \
171         mkdir -p /poet-shared/validator-4 && \
172         cp -a /etc/sawtooth/keys /poet-shared/validator-4/ && \
173         sawtooth-validator -v \
174               --bind network:tcp://eth0:8800 \
175               --bind component:tcp://eth0:4004 \
176               --bind consensus:tcp://eth0:5050 \
177               --peering static \
178               --endpoint tcp://validator-4:8800 \
179               --peers tcp://validator-0:8800,tcp://validator-1:8800,tcp://validator-2:8800,tcp://validator
                      -3:8800 \
180               --scheduler parallel \
181               --network-auth trust
182       "
183     environment:
184       PYTHONPATH: "/project/sawtooth-core/consensus/poet/common:\
185         /project/sawtooth-core/consensus/poet/simulator:\
186         /project/sawtooth-core/consensus/poet/core"
187     stop_signal: SIGKILL
188
189   rest-api-0:
190     image: hyperledger/sawtooth-rest-api:chime
191     container_name: sawtooth-rest-api-default-0
192     expose:
193       - 8008
194     command: |
195       bash -c "
196         sawtooth-rest-api \
197           --connect tcp://validator-0:4004 \
198           --bind rest-api-0:8008
199       "
200     stop_signal: SIGKILL
201
202   rest-api-1:
203     image: hyperledger/sawtooth-rest-api:chime
204     container_name: sawtooth-rest-api-default-1
205     expose:
206       - 8008
207     command: |
208       bash -c "
209         sawtooth-rest-api \
210           --connect tcp://validator-1:4004 \
211           --bind rest-api-1:8008
```

```
212         "
213       stop_signal: SIGKILL
214
215   rest-api-2:
216       image: hyperledger/sawtooth-rest-api:chime
217       container_name: sawtooth-rest-api-default-2
218       expose:
219         - 8008
220       command: |
221         bash -c "
222           sawtooth-rest-api \
223             --connect tcp://validator-2:4004 \
224             --bind rest-api-2:8008
225         "
226       stop_signal: SIGKILL
227
228   rest-api-3:
229       image: hyperledger/sawtooth-rest-api:chime
230       container_name: sawtooth-rest-api-default-3
231       expose:
232         - 8008
233       command: |
234         bash -c "
235           sawtooth-rest-api \
236             --connect tcp://validator-3:4004 \
237             --bind rest-api-3:8008
238         "
239       stop_signal: SIGKILL
240
241   rest-api-4:
242       image: hyperledger/sawtooth-rest-api:chime
243       container_name: sawtooth-rest-api-default-4
244       expose:
245         - 8008
246       command: |
247         bash -c "
248           sawtooth-rest-api \
249             --connect tcp://validator-4:4004 \
250             --bind rest-api-4:8008
251         "
252       stop_signal: SIGKILL
253
254   control-tp-0:
255       image: control-tp-python-local:latest
256       container_name: sawtooth-control-tp-python-default-0
257       expose:
258         - 4004
259       command: control-tp-python -vv -C tcp://validator-0:4004
260       stop_signal: SIGKILL
261
262   control-tp-1:
263       image: control-tp-python-local:latest
264       container_name: sawtooth-control-tp-python-default-1
```

```
    expose:
      - 4004
    command: control-tp-python -vv -C tcp://validator-1:4004
    stop_signal: SIGKILL

  control-tp-2:
    image: control-tp-python-local:latest
    container_name: sawtooth-control-tp-python-default-2
    expose:
      - 4004
    command: control-tp-python -vv -C tcp://validator-2:4004
    stop_signal: SIGKILL

  control-tp-3:
    image: control-tp-python-local:latest
    container_name: sawtooth-control-tp-python-default-3
    expose:
      - 4004
    command: control-tp-python -vv -C tcp://validator-3:4004
    stop_signal: SIGKILL

  control-tp-4:
    image: control-tp-python-local:latest
    container_name: sawtooth-control-tp-python-default-4
    expose:
      - 4004
    command: control-tp-python -vv -C tcp://validator-4:4004
    stop_signal: SIGKILL

  settings-tp-0:
    image: hyperledger/sawtooth-settings-tp:chime
    container_name: sawtooth-settings-tp-default-0
    expose:
      - 4004
    command: settings-tp -v -C tcp://validator-0:4004
    stop_signal: SIGKILL

  settings-tp-1:
    image: hyperledger/sawtooth-settings-tp:chime
    container_name: sawtooth-settings-tp-default-1
    expose:
      - 4004
    command: settings-tp -v -C tcp://validator-1:4004
    stop_signal: SIGKILL

  settings-tp-2:
    image: hyperledger/sawtooth-settings-tp:chime
    container_name: sawtooth-settings-tp-default-2
    expose:
      - 4004
    command: settings-tp -v -C tcp://validator-2:4004
    stop_signal: SIGKILL

```

```
318   settings-tp-3:
319     image: hyperledger/sawtooth-settings-tp:chime
320     container_name: sawtooth-settings-tp-default-3
321     expose:
322       - 4004
323     command: settings-tp -v -C tcp://validator-3:4004
324     stop_signal: SIGKILL
325
326   settings-tp-4:
327     image: hyperledger/sawtooth-settings-tp:chime
328     container_name: sawtooth-settings-tp-default-4
329     expose:
330       - 4004
331     command: settings-tp -v -C tcp://validator-4:4004
332     stop_signal: SIGKILL
333
334   poet-engine-0:
335     image: hyperledger/sawtooth-poet-engine:chime
336     container_name: sawtooth-poet-engine-0
337     volumes:
338       - poet-shared:/poet-shared
339     command: "bash -c \"\
340         if [ ! -f /poet-shared/poet-enclave-measurement ]; then \
341             poet enclave measurement >> /poet-shared/poet-enclave-measurement; \
342         fi && \
343         if [ ! -f /poet-shared/poet-enclave-basename ]; then \
344             poet enclave basename >> /poet-shared/poet-enclave-basename; \
345         fi && \
346         if [ ! -f /poet-shared/simulator_rk_pub.pem ]; then \
347             cp /etc/sawtooth/simulator_rk_pub.pem /poet-shared; \
348         fi && \
349         while [ ! -f /poet-shared/validator-0/keys/validator.priv ]; do sleep 1; done && \
350         cp -a /poet-shared/validator-0/keys /etc/sawtooth && \
351         poet registration create -k /etc/sawtooth/keys/validator.priv -o /poet-shared/poet.batch && \
352         poet-engine -C tcp://validator-0:5050 --component tcp://validator-0:4004 \
353     \""
354
355   poet-engine-1:
356     image: hyperledger/sawtooth-poet-engine:chime
357     container_name: sawtooth-poet-engine-1
358     volumes:
359       - poet-shared:/poet-shared
360     command: "bash -c \"\
361         while [ ! -f /poet-shared/validator-1/keys/validator.priv ]; do sleep 1; done && \
362         cp -a /poet-shared/validator-1/keys /etc/sawtooth && \
363         poet-engine -C tcp://validator-1:5050 --component tcp://validator-1:4004 \
364     \""
365
366   poet-engine-2:
367     image: hyperledger/sawtooth-poet-engine:chime
368     container_name: sawtooth-poet-engine-2
369     volumes:
370       - poet-shared:/poet-shared
```

```
    command: "bash -c \"\
        while [ ! -f /poet-shared/validator-2/keys/validator.priv ]; do sleep 1; done && \
        cp -a /poet-shared/validator-2/keys /etc/sawtooth && \
        poet-engine -C tcp://validator-2:5050 --component tcp://validator-2:4004 \
    \""

  poet-engine-3:
    image: hyperledger/sawtooth-poet-engine:chime
    container_name: sawtooth-poet-engine-3
    volumes:
      - poet-shared:/poet-shared
    command: "bash -c \"\
        while [ ! -f /poet-shared/validator-3/keys/validator.priv ]; do sleep 1; done && \
        cp -a /poet-shared/validator-3/keys /etc/sawtooth && \
        poet-engine -C tcp://validator-3:5050 --component tcp://validator-3:4004 \
    \""

  poet-engine-4:
    image: hyperledger/sawtooth-poet-engine:chime
    container_name: sawtooth-poet-engine-4
    volumes:
      - poet-shared:/poet-shared
    command: "bash -c \"\
        while [ ! -f /poet-shared/validator-4/keys/validator.priv ]; do sleep 1; done && \
        cp -a /poet-shared/validator-4/keys /etc/sawtooth && \
        poet-engine -C tcp://validator-4:5050 --component tcp://validator-4:4004 \
    \""

  poet-validator-registry-tp-0:
    image: hyperledger/sawtooth-poet-validator-registry-tp:chime
    container_name: sawtooth-poet-validator-registry-tp-0
    expose:
      - 4004
    command: poet-validator-registry-tp -C tcp://validator-0:4004
    environment:
      PYTHONPATH: /project/sawtooth-core/consensus/poet/common
    stop_signal: SIGKILL

  poet-validator-registry-tp-1:
    image: hyperledger/sawtooth-poet-validator-registry-tp:chime
    container_name: sawtooth-poet-validator-registry-tp-1
    expose:
      - 4004
    command: poet-validator-registry-tp -C tcp://validator-1:4004
    environment:
      PYTHONPATH: /project/sawtooth-core/consensus/poet/common
    stop_signal: SIGKILL

  poet-validator-registry-tp-2:
    image: hyperledger/sawtooth-poet-validator-registry-tp:chime
    container_name: sawtooth-poet-validator-registry-tp-2
    expose:
      - 4004
```

```
424       command: poet-validator-registry-tp -C tcp://validator-2:4004
425       environment:
426         PYTHONPATH: /project/sawtooth-core/consensus/poet/common
427       stop_signal: SIGKILL
428
429     poet-validator-registry-tp-3:
430       image: hyperledger/sawtooth-poet-validator-registry-tp:chime
431       container_name: sawtooth-poet-validator-registry-tp-3
432       expose:
433         - 4004
434       command: poet-validator-registry-tp -C tcp://validator-3:4004
435       environment:
436         PYTHONPATH: /project/sawtooth-core/consensus/poet/common
437       stop_signal: SIGKILL
438
439     poet-validator-registry-tp-4:
440       image: hyperledger/sawtooth-poet-validator-registry-tp:chime
441       container_name: sawtooth-poet-validator-registry-tp-4
442       expose:
443         - 4004
444       command: poet-validator-registry-tp -C tcp://validator-4:4004
445       environment:
446         PYTHONPATH: /project/sawtooth-core/consensus/poet/common
447       stop_signal: SIGKILL
```